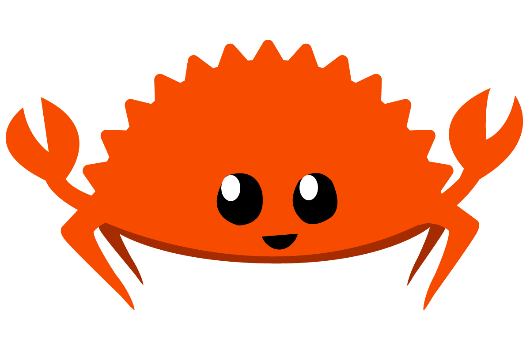
# Rust

Last Updated: July 2022 v1.62.0

1. About Rust:
   1. Fast, multi-threaded, type-safe & memory efficient language that eliminates most bugs at compile-time. It also bridges the gap between a low-level language and a high-level language with its compiler. It uses rustup to maintain versions on platforms and cargo which is its package manager. The file ext. for rust files is \*.rs. Its mascot is called Ferris, a red crusty crab and its users, Rustaceans.
   2. Default formatter shenanigans:
      1. The names should follow snake\_case and not camelCase or PascalCase. This is applied to all names, folder names, file names, variables, functions etc. However, it is not applied to things like type names where we need to follow CamelCase.
   3. In rust, lines need to be ended with ;.
   4. A rust project needs a cargo.toml file at the bare minimum. We can create a new project with cargo new <folder/projectname>. We can then build the project with cargo build. And then run it with cargo run. A cargo.toml file is a cargo configuration file for the project, it looks like so

[package]

name = "citorust\_alpha"

version = "0.1.0"

edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

toml stands for Tom’s Obvious, Minimal Language.

Cargo projects, by default, needs their .rs files inside an src folder, this folder must be at the same level as the cargo.toml file.

* + 1. Some quick commands:

cargo build: builds the project and creates an executable in debug/ . Use --release flag to build in release mode. Diff b/w debug build and release build is that debug build is compiled faster but runs slower whereas release compiles slower but runs faster.

cargo run: builds the project or skips it if there already exists a latest build and then runs the exe.

cargo check: checks the project by building it but not creating an exe, faster than cargo build.

* + 1. Repo: crates.io is the Rust community’s central crate repository.
    2. To add a crate to our project,

Find the crate version and name on crates.io

Add

<packge-name>=”<version>” to the cargo.toml

Then open the items of the package in the scope with ‘use’.

* + 1. The standard library crates are already present in most Rust installations, we don’t need to specify its crates in the cargo.toml but we do need to open the items in the scope (using ‘use’) for easy access.
  1. fn main() {…} in any .rs file is the entrypoint for any rust project.
  2. For individual files, we compile them with rustc <filename>.rs. It generates an \*.exe in windows and that can be directly run with ./filename.exe. Regardless of the approach used to generate an executable, it is always AOT (ahead-of-time) compiled, meaning, if we pass the executable to another system, it does not need to have Rust installed in order to run it. Whereas \*.py or \*.js files need Python or Javascript installed in order to be run.

1. Variables: Variables in rust are immutable by default, meaning their value can’t be updated after initializing it once.

For ex.:

let x: i32;

x=6;

//will work

//but if we do

x=7

//then this will throw an error.

* 1. But immutability is only the default, it’s not the only choice.

To create mutable variables,

let mut <varname>:<var type>;

While mutable variables allow reassignment of value, they forbade the reassignment of types so the variable can only get a value of the same type.

* 1. Constants: Declare constant variables with

const <varname>:<type>= <value>;

Unlike variables created with let, const vars are always immutable. Another feature of const is that it allows constant expressions like C++. So

const ABC: u32 = 2\*2\*3+3;

is valid.

* 1. Shadowing: In rust, we can create a new variable with the same name as a previous one. The new variable ‘overshadows’ the older one and will be accessed when called later in the scope instead of the variable being shadowed. The overshadowing variable won’t be accessed if not in present or parent scope. This allows us to use some cool things like

let spaces = " ";

let spaces = spaces.len();

The first spaces is of type string while the latter is a number type.

* 1. Data types: Rust is a statically typed language, meaning the type of values must be resolved at compile time. Explicitly specifying a type isn’t required as the compiler can infer the type of a variable automatically. It is a bit more powerful than other languages as type can be inferred even after the declaration of variable.

However, types must be specified when multiple types are possible from an expression. For ex.

#![allow(unused)]

fn main() {

let guess: u32 = "42".parse().expect("Not a number!");

}

works as we specify the type. But if we remove the type then it throws an error. This is to say, the parse() method picks up the type given to the variable. The compiler passes the type given to the receiver to the method for its generics, so parse knows it has to work for type u32.

* + 1. NNBD: Rust is non-nullable by default, infact even the concept of null doesn’t exist in Rust. Every variable must either have a value or a type defined at-least and every variable must be assigned a value before being used. All of this is checked during compile time itself, and after compilation a variable must have a type.

For ex.:

let y;

// error as y doesn’t have a type

let y: i32;

//ok

let y;

y=2;

//ok and y’s type is inferred as i32.

let y:i32;

let x= y;

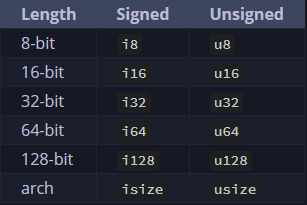
//error as y doesn’t have a value.

let y=2;

//ok and y’s type is inferred as i32.

* + 1. 2 basic data types: Scalar and Compound.
       1. Scalar: A type that represents a single value. It has

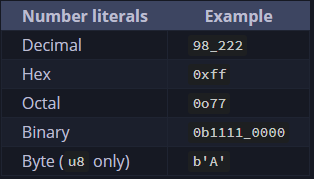
integers: Numbers without fractional components.



Signed ints are the ints that care for the sign, + or -. These can store -ve as well as +ve values. Stored using 2’s complement representation. –(2n-1) < Value range < 2n-1. Unsigned ints can only be +ve. Value range < 2n-1 where n is the no. of bits.

The isize and usize are used to denote size of collections and depend on the architecture of the program, i.e. 64 bits on x64 and 32 bit on x86.

The values theirselves can denote their types as well.



or in the form 57u8 (57 of type u8).

floating-point nums: f32 and f64, 32 bits and single precision & 64 bits and double precision respectively. In modern CPUs, f64 is as fast as f32 but with greater precision, which is why if f32 isn’t explicitly specified, a floating number is assumed as f64 in rust.

bools: true or false.

chars: A character in rust is a 4 byte Unicode Scalar Value stored inside ‘ ‘ (and not “ “, which is used for strings). Since it is Unicode, it can store ASCII but also other language characters and even emojis, in the range U+0000 to U+D7FF and U+E000 to U+10FFFF inclusive. So a value ‘❤️’ is a single char in rust.

* + - 1. Compound Types: Groups multiple values into a single type. Rust has Tuples and Arrays as compound types.

Tuples: Fixed-size, multi-type and immutable.

For ex.:

let tup: (u8, bool, f32)= (1, true, 2.0);

and to access the values,

let x =tup.0;

i.e., access values by their indices. This is called tuple indexing.

Arrays: Fixed-size, single-type & immutable.

For ex.:

let arr: [i32;3]= [1,2,3];

and access with

let x = arr[0];

Arrays can also be initialized with default values,

let arr = [3;5];

arr will have size 5 and the values in all those 5 indices will be 3.

Defining types is optional in compound types.

If we specify an index out of bounds for either type, it will give an error at compile time, and if the value is generated at run-time then the program will panic and exit at that point without proceeding forward.

* + 1. Overflow and prevention: In rust, if an integer overflows (value greater than the size allowed by type) then, if it’s in debug mode then the program panics, and if it’s in release mode then the value wraps around (new value = overflowing value – size of type, if it’s still overflowing then repeat). To allow better checks against said overflows, rust provides a few standard methods available in same named crates.
       1. Wrap in all modes with the wrapping\_\* methods, such as wrapping\_add
       2. Return the None value if there is overflow with the checked\_\* methods
       3. Return the value and a boolean indicating whether there was overflow with the overflowing\_\* methods
       4. Saturate at the value’s minimum or maximum values with saturating\_\* methods
    2. Unit type: An empty tuple is called a unit. It’s like (). Every function returns a unit if it doesn’t return any other value.
    3. Destructuring: Using pattern-matching, we can destructure compound types into scalar types. Just like structured bindings in c++.

For tuples:

let tup =(1,’a’,true);

let (x,y,z) = tup;

For arrays:

let arr= [1,2,3];

let [x,y,z] = arr;

* + 1. Collections: These types are generally on the heap and are provided by the standard library. They store collections of values on heap, which allows them to have unfixed size and grow dynamically. But it also means these data types take time in (de)allocation.
       1. Some collection types:
          1. Vector: Just like vectors in C++. Contiguous blocks of dynamically allocated memory with same data types.

To create empty new vector,

let vec\_x: Vec<i32> = Vec::new();

Here defining the type is necessary as that is specifying the type for the generic Vec<T> used in Vec::new(), however Rust can infer the type if the variable is modified later. This is where we can see the powerful type inference in action.

We can use the macro vec! for simpler syntax,

let vec\_x= vec![1,2,3];

To push elements to it,

let mut vec\_x= Vec::new();

vec\_x.push(<value>);

and Rust will pick up the type of value and apply it to vec\_x automatically.

When a vector goes out of scope, all its elements are dropped too. That is when the owner of a vector is dropped, all elements follow.

Reading element: To access an element, vectors can use 2 ways,

let arr=vec![1,2,3];

let elem= &arr[2];

//or

let elem2= match arr.get(2) {

Some(value)=> value,

None => 0,

}

.get method returns an Option<&T> and allows for checking if value is in index. The former way causes the program to panic on bad index.

We get the normal copy/move behavior as defined later in the doc.

When we access elements by reference, the same mutability/immutability rules are enforced. So,

let mut vec1= vec![1,2,3];

let elem= &vec1[2];

vec1.push(2);

is an error. As push borrows a mutable reference of the vector and we already have an immutable reference.

To read entire vector,

for i in &vec1 {

…

}

and to read and edit the vector,

for i in &mut vec1 {

\*i += 2;

}

We use the dereference operator here as unlike normal operations it is not assumed automatically.

* + - * 1. String: Rust has a bit complex string data type. Firstly, strings are implemented as collection of bytes and this collection provides a lot of useful methods to work with said bytes, such as bytes being interpreted as text, manipulation etc.

There’s only 1 type of string at the core of Rust, str. An str is a string slice. A string literal, i.e., compile-time string is a string slice but it is stored in the program’s binary.

A String data type is a growable, mutable and owned collection in the stdlib. There’s some complexity involved in converting between String and str type.

Both of these types are UTF-8 encoded

To create a string:

let x= “abc”.to\_string();

let mut y=String::new();

y=”abc”.to\_string();

let z= String::from(“abc”);

All do the same thing.

Appending to string: There are several ways,

let mut x=”abc”.to\_string();

x.push\_str(“yoo”);

x.push(‘y’);

x+=”yoo”;

We can’t add 2 String types together, but we can add str to String. Similarly we can’t add to an &str as it is an immutable reference of an str slice, we can modify it but not add another &str.

We can add String and &String though,

let a= String…

let b=…

let c= a+ &b;

The add func is defined like so,

fn add(self, s: &str) -> String {…}

So while a loses ownership, b holds its ownership and a copy of b’s value is appended to a then a’s ownership is passed back to c.

Here, Rust compiler automatically coerces &String into an &str (&s2 to &s2[…]), this is known as deref coercion.

We can add as many &str or &String but we need a single String in the expression.

Lastly we can use the format! macro expression,

let c = format!(“{} yo {}”, a,b);

format! returns a String, and it uses references for all its parameters automatically and so both a and b remain valid while both get copied into a String whose ownership is returned to c.

String indexing: It is not possible using integers in Rust, i.e.,

let x = “as”;

let y= x[1]; //is an error.

This is because Rust’s String is internally a wrapper over Vec<u8> (byte vector) and the String type is UTF-8 encoded.

For ex.:

let hello = String::from("Здравствуйте");

Here hello’s length is not 12, but 24, because encoding each letter here into UTF-8 takes 2 bytes, that is, 2 bytes represent each visible letter here.

hello[0] would mean byte at address 0 of vector, which is a 3. However, the 3 is actually a part of the Cyrillic letter Ze and not Arabic number 3 and needs 2 bytes to be properly represented. This is a source of confusion hence why Rust doesn’t allow String indexing.

Chars in String:

There are 3 ways Rust sees characters in String.

For “नमस्ते”,

It’s length is 18 in bytes and this is how it looks,

[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]

Rust sees chars as Unicode Scalar Values, basically like this

['न', 'म', 'स', '्', 'त', 'े']

The 4th and 6th letter here are called diacritics as they don’t make sense on their own.

Lastly, here’s how we interpret these letters,

["न", "म", "स्", "ते"]

These are the grapheme clusters of the word Namaste.

A grapheme cluster is a more accurate name of what we call a letter in English.

This is why indexing is not done in Rust. This is also why, slice operations are risky, for ex.:

let hello = String::from("Здравствуйте");

hello[0..1];

will cause the program to panic as the first byte is a part of 2 to make up a Unicode Scalar value, so hello[0..2] would work.

Iterating over String values: Due to the complex nature of Unicode, there are multiple ways to iterate over strings in Rust.

for elem in hello.chars() {

…

}

gets the Unicode Scalar values.

for elem in hello.bytes() {

…

}

gets the bytes.

String.as\_string(): To get an &str from a String object.

Raw string: Get a string as defined, we use “\ … “

For ex.:

let contents = "\

Rust:

safe, fast, productive.

Pick three.

Duct tape.";

or use r# and #

let abc= r# “my string”#;

* + - * 1. HashMap: Stores K,V pairs of objects where each Key corresponds to a Value. It’s like a map, but it runs a hash function over the values. The hash function is a SipHash which isn’t the fastest hasher but is very secure.

For ex.:

use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);

Just like Vectors, hashmaps store their data on the heap. They also have the same data type for their keys and the same type for their values.

We don’t have HashMap in the prelude and they also don’t have a macro. Still, we can combine 2 vectors into a HashMap using iterators. For ex.:

use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];

let initial\_scores = vec![10, 50];

let mut scores: HashMap<\_, \_> =

teams.into\_iter().zip(initial\_scores.into\_iter()).collect();

zip takes 2 iterators and pairs them into a tuple and collect collects the tuples and then returns a required type which is HashMap here. Since collect needs a type to know what to return to, and HashMap needs types to define K and V, Rust allows us to use \_ as placeholder types. Then it infers the correct types.

References: HashMaps follow the same copy/move rules. So,

let a = String…

let mut b= HashMap::new();

b.insert(“yo”, a);

will work but a’s ownership will be passed to b. If we had used &a, however, then that wouldn’t be the case.

Accessing values:

let mut a= HashMap…

let c= String::from(“abc”);

let b= a.get(&c);

We pass &c to preserve ownership. It will get an Option<&V> into b, if the key is found then Some’s the value else None.

Or

with a loop

for (key,value) in &a {…}

Inserting: Using the insert function. If a key exists then its value is overwritten. If we want to check before inserting we can use the entry method,

For ex.:

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);

let insertedValue= scores.entry(String::from("Yellow")).or\_insert(50);

Here entry returns an enum Entry which defines if the key exists or not. The Entry enum also defines a few methods and one of them is or\_insert, which inserts the key and the value if the key doesn’t exist otherwise does nothing. Either way, or\_insert returns a mutable reference of the value to the key.

* + 1. Type Alias: We can declare type aliases in Rust. To do so,

type Yo = i32;

let x: Yo = 2;

and it works. This syntax looks like associated types because it is, except the type is resolved right away.

* + 1. Empty Type: Aka Never Type in Rust. It is represented with ! and it has no values. So it is like void in C++. By default functions return an empty tuple, so it is used like

fn yo() -> ! {…}

Although a syntax error, it says function yo returns never. This type of function is called a diverging function as control never returns from this function. {} blocks always return something.

The never type has some uses, namely, it tells Rust that a given piece of code never returns hence the only possible return types are others. For ex.:

let x= Some(2);

let y= match x {

Some(\_) => 2,

None => “ye”

};

is an error as match needs to return the same type for all arms, however,

let y= match x {

Some(\_) => 2,

None => panic!(…)

};

is not an error. This is because Rust knows if the never type is returned then the control flow has changed and hence it doesn’t need to worry about the arm’s type. This is also to say, if a function or macro never returns then the program will go into a different place and the control won’t ever come back to finish the expression. continue also returns a never type.

* + 1. Dynamically Sized Types (DSTs): aka unsized types are types whose size can only be known at runtime. Like str is a DST.

For ex.:

let x: str= “yo”;

let y: str= “aaaaa”;

Both take different amount of memory, but Rust needs to know exactly how much memory a type needs and all values of a type must be the same size. This is why above example is an error.

So to solve it,

let x: &str =…;

let y: &str =…;

works. This is because unlike raw types, references need to store only some metadata, a reference on str is called a slice and a slice type stores the initial character’s position and the total length of the string. Both of which are deterministic and the total size of the &str can be known at compile time.

Similarly all other DSTs work and we can say DSTs internally use a pointer of some kind. Traits are DSTs too as their size isn’t known at compile time. This is why trait objects need to be defined with &dyn Trait or dyn Trait.

Since DSTs are a core concept Rust actually defines non DSTs with Sized trait, this trait is applied to all primitive types and defines their size at compile time, and since it is a trait, it is implicitly present in all generic definitions too.

* + 1. Size of: To get the size of types, we can use size\_of

For ex.:

core::mem::size\_of::<i32>();

* 1. Casting: Generally types have a From and an Into method to convert to and from most types. But there’s a more primitive alternative, ‘as’ keyword. It can cast primitive types into other primitives. For ex.:

let thing1: u8 = 89.0 as u8;

assert\_eq!('B' as u32, 66);

assert\_eq!(thing1 as char, 'Y');

let thing2: f32 = thing1 as f32 + 10.5;

assert\_eq!(true as u8 + thing2 as u8, 100);

are all true.

let x = 2 as i32;

works too, however it is better (for performance) to use let x: i32 = 2;

In this case, if there is a cast needed then the appropriate method on the type is automatically called.

* 1. static type: Global variables are declared with static keyword. In Rust, we can’t declare global variables with let, we have to use static which also means the value can’t change in safe Rust (it is allowed in unsafe rust).

For ex.:

static HELLO\_WORLD: &str = "Hello, world!";

as for mutable ones,

static mut x: … = …

Global variables are allowed to reference other global variables (but only immutably). They always need to specify the type as type inference doesn’t apply to them.

1. Scopes: A scope is the range within a program for which an item is valid. There’s global scopes and local scopes. A new custom scope can be created with {…}. So

fn xyz() {

{

let y:i32=2;

}

println!(y); //will throw error

}

has 2 scopes, and y is in the custom inner scope.

* 1. Scopes are an important concept in understanding ownership rules, expressions, statements etc.

1. Functions: Just like other languages. Unlike C++, Functions can be defined anywhere, but they must be accessible by the caller’s scope.

Basic syntax:

fn <function name>(<params>)-> <return type> {

…

return <value>;

}

For ex.:

fn another\_function(x: i32, unit\_label: char) {

println!("The value of x is: {x} and char is {unit\_label}");

}

* 1. Parameters/Arguments: If we pass concrete values to a function, then they are called the arguments, whereas parameters are the types that the function accepts and defines in its function signature. Functions must define the types of its params.
  2. Statements vs Expressions: Statements are instructions that perform some actions and don’t return a value. Expressions are the same but return a value. Functions definitions in Rust are statements. But calling a function is an expression. This distinction is not as visible in other languages but in rust,

let x = (let y = 6);

is an error as ‘let’ is a statement and hence doesn’t return anything.

However, a custom scope block is an expression. So,

let y = {

let x= 1;

x+1

};

println!(“{}”,y); //prints 2

works. And x+1 is the line being returned. If x+1 had a ; then it’d have become a statement and hence the block would have returned a Unit instead. Also note the semicolon at the end of the scope block, it is necessary if the scope block’s return is required.

* 1. Return type: If we don’t specify a return type then it is assumed to be Unit. If we do specify, as is required for all types, we can return values in 2 ways, as expressions or with return syntax.

For ex.:

fn abc() -> i32 {

2\*2

}

works but 2\*2; would fail as the expression is calculated then and there and turns into a statement. Then the function is returning a Unit whereas it has specified i32 so it is an error.

Alternatively,

fn abc() -> i32 {

return 2\*2;

}

can be used. return runs an expression and passes its value.

* 1. Rust doesn’t allow operator overloading. However we can define operator behavior using traits, they are defined in std::ops crate. For ex.:

use std::ops::Add;

struct X{…}

impl Add for X {

type Output= i32;

fn add(&self, other: X) -> i32 {

…

}

}

let x= X{…};

then x+x works.

* 1. Diverging function: A function that returns never (! type).
  2. Function Pointers: Functions can be passed around just like closures. Functions coerce into the fn type. To use them

fn yo(x: i32, y: &str) -> bool {

return true;

}

fn takeYo(aFn: fn(i32, &str) -> bool, b: i32) {

let ans = aFn(b, "naa");

}

fn main() {

let x = takeYo(yo, 2);

}

works. Passes yo’s address to aFn. Function pointers implement all 3 closure traits (Fn, FnMut and FnOnce) so a function pointer can always be passed to a function that needs a closure.

1. Comments work as normal, // for single line and /\* \*/ for multi-line. /// for documentation is supported as well.
   1. Documentation: /// for documentation comments. /// supports Markdown so HTML formatting is supported in them.

For ex.:

/// Adds one to the number given.

///

/// # Examples

///

/// ```

/// let arg = 5;

/// let answer = my\_crate::add\_one(arg);

///

/// assert\_eq!(6, answer);

/// ```

pub fn add\_one(x: i32) -> i32 {

x + 1

}

* + 1. cargo doc: Running this generates the HTML documentation for all the doc commented lines by using the rustdoc tool distributed with Rust. –open flag on it will open the built HTML page.
    2. //!: These type documentation comments document the parent item of a doc comment, i.e., the item that contains the doc comment following the //! styled doc comment. For ex.:

//! # My Crate

//!

//! `my\_crate` is a collection of utilities to make performing certain

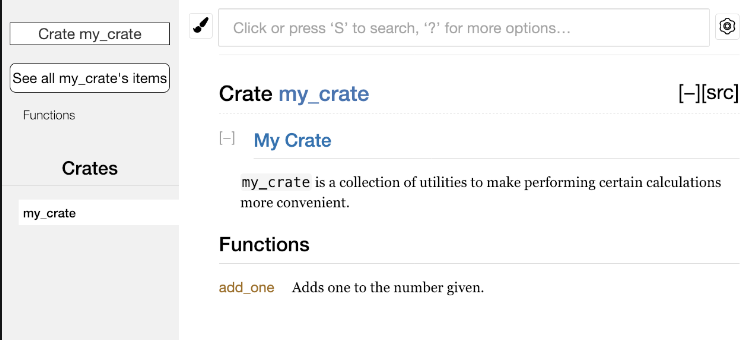
//! calculations more convenient.

/// Adds one to the number given.

// --snip--

//! documents the crate. This is because the next doc comment’s parent is the crate itself,i.e. /// is in a crate and //! adds documentation for it.

This will generate this documentation,



1. Control Flow: Unlike languages like JS, non-booleans aren’t automatically converted to Booleans.
   1. Conditionals:
      1. If-else: The if/else blocks are called ‘arms’.

if x< y {…}

else if x> y {…}

else {…}

* + - 1. if is an expression. So it returns a value. Hence we can do this,

let z = if 2<3 { 2 } else { 3 };

However, if we return values of differing types then it is an error as both arms have to return values of same type and that type must be assignable.

* + 1. match: Just like switch case. match is an expression so it can return values. The arms of match accept patterns as conditions. Just like Kotlin switch case, the break; is implicit in the arms of match. The match arms have 2 parts, pattern and code, if the pattern matches (going from top to bottom in arm list), then the code is ran and the value returned. However, match arms need to be exhaustive, i.e., all possible cases need to be covered by the arms.

For ex.:

let value = 2;

let yo= match value {

3 => “yo”.to\_string(),

2 => “ye” .to\_string(),

other => “na” .to\_string()

}

We can use a variable as a pattern to catch all other values, this is like ‘default’ case. In this case other is a variable and it takes the value from value variable that has failed to match with other arms. We can use the value in other in its body.

Alternatively, we can use a special letter, ‘\_’ to match-all other cases, but unlike the variable match-all this symbol doesn’t take the value in value variable.

If we don’t desire the pattern to do anything we can use the unit,

match x {

1 => (),

2 => “yee”

}

match’s actual syntax is

match VALUE {

PATTERN => EXPRESSION,

PATTERN => EXPRESSION,

…

}

* + - 1. Match Guard: A match arm can contain another condition using an if. This allows it to have more complex conditionals.

For ex.:

let a= Some(2);

let y=true;

match a {

Some(x) if x==3 =>…

Some(4) | Some(5) | Some(6) if y=>…

Some(x) =>…

\_ =>…

}

works. The pattern is like (condition 1) if condition 2

so like (Some(4) | Some(5) | Some(6)) if y

* + 1. if let: This syntax allows us to just define a single match arm in a way of reducing boilerplate with match.

For ex.:

For this match

let some= Some(24);

match some {

Some(value) => println!(“yo”),

\_ => (),

};

We don’t care about the None value, so we can simply write the single arm Some(value) as

if let Some(value) = some {

…

}

Both the approaches are doing the same thing, but with if let we reduce boilerplate and compiler automatically understands it’s a match arm with \_ => () handling rest of the cases.

There’s also if let else, the else arm let’s us define the catch-all \_ block,

So the same code can be written as,

if let Some(value) = some {…}

else {

…

}

And now this covers a given arm plus the catch all expression.

if let also has else if, so

fn main(){

let a=Some(2);

let b= Some("yee");

if let Some(value)= a {

println!("{}",value);

}

else if let Some(value)= b {

println!("{}",value);

}

else {

println!("naaa");

}

}

is a perfectly fine code. If let allows us to match any valid pattern with any valid value unlike match which only allows a single value to be compared against.

* 1. Loops: loop, while and for. break and continue are also expressions, and can be used with loops for some neat operations.
     1. loop: Runs an infinite loop. This is an expression.

For ex.:

let mut x= 1;

let z = loop {

if x==20 {

break 25

}

x+=1;

};

break can return a value as well. Continue can’t. Both can optionally use ; at the end.

* + - 1. Loop Label: We can use this to break/continue a specific loop.

For ex.

let z = 'xyz\_loop: loop {

if x==20 {

'two: loop {

if x==25 {

break 'two;

}

x+=1;

}

break 'xyz\_loop x;

}

x+=1;

};

* + 1. while: Similar to other languages.

while x>2 {

…

}

There’s also while let,

let mut stack = Vec::new();

stack.push(1);

stack.push(2);

stack.push(3);

while let Some(top) = stack.pop() {

println!("{}", top);

}

* + 1. for loop:

let arr = [1,2,5,3]

for element in arr {

…

}

* + - 1. We can use for loop with Range too, this is a type provided by the standard lib. As it is in standard lib, we don’t need to import any crate or anything.

For ex.:

for elem in (1..4).rev() {…}

1..4 generates a Range with 3 elements 1, 2, 3. Here rev() reverses the range.

* + - 1. for with index: If we need the index too, we can use tuple destructuring

for (index, elem) in arr.iter().enumerate() {…}

iter is already defined for [T;KSize] types and enumerate() on it returns a tuple with index and value.

* 1. Patterns: They are a special syntax that allow Rust to perform matching against structures of types, both, simple and complex. Patterns allow a value to be compared against a special syntax and hence alter the flow of the program.

For ex.:

\_ is a pattern that matches anything,

match x {

\_ => println!(“yo”);

}

\_ will catch all values.

* + 1. Patterns are everywhere, even let x=2; is let PATTERN = EXPRESSION;

where it says, bind whatever expression is to the given pattern.

This is why,

let (x, y, z) = (1, 2, 3);

works.

Even functions take patterns as params,

fn poopoo(&(x, y): &(i32, i32)) {

println!("Current location: ({}, {})", x, y);

}

let pt = (1,2);

poopoo(&pt);

works. This applies to methods and closures too.

* + 1. Refutable Patterns: Patterns that will match no matter the value are known as irrefutable patterns, like let x= 5;

But patterns that might fail to match, like if let Some(value)=aVariable {…}

are refutable as they might fail, like in this case when aVariable is a None rather than Some.

Function params, let declarations and for loops can only take irrefutable patterns.

For ex.:

let Some(x) = some\_option\_value;

will fail to compile as an irrefutable pattern is needed on the LHS.

Irrefutable pattern is also what the last match arm needs as its pattern.

* + 1. Types of patterns:
       1. Concrete values:

For ex.:

match x{

1 => “ye”,

\_ => “na”,

}

* + - 1. Named variables: External variables are shadowed in an expression of a pattern.

For ex.:

let x= Some(2);

let y=3;

match x {

Some(20) =>…,

Some(y) =>println!(“{}”,y),

\_ => “baaaa”,

};

Here, the 2nd arm matches. The first arm checks Some(50) == Some(2) and fails, the 2nd arm checks Some(T) to x and passes and then puts the value in y which shadows the external y and hence it prints 2.

* + - 1. OR: To match multiple patterns with an OR we use single | .

For ex.:

match x {

1 | 2 => …,

1 | 2 | 3 => …

}

Any number of ORs can be used.

* + - 1. Range of values: We use ..

For ex.:

match x {

1..5 => …

}

checks if x has a value between 1(inclusive) and 5 (exclusive). Exclusive range end is an experimental feature right now. To have an inclusive range end,

1..=5

char ranges are accepted too, so ‘a’..=’z’ works too.

* + - 1. Destructuring:

For structs:

struct AB{

x:i32,

y:i32,

}

let p= AB {x: 1, y:2};

let AB {x: a, y: b} = p;

works, puts x of p in a and y of p in b.

Similarly,

let AB {x, y} =p;

puts x of p in x and y of p in y.

We can use this pattern for matching as well,

match p {

AB {x , y: 2} => …

AB {x ,y} => …

}

arm 1 says if y of p is 2 then match and x can be anything. It also stores x and y in x and y.

For enums,

enum Message {

Quit,

Move { x: i32, y: i32 },

Write(String),

ChangeColor(i32, i32, i32),

}

fn main() {

let msg = Message::ChangeColor(0, 160, 255);

match msg {

Message::Quit => {

println!("The Quit variant has no data to destructure.")

}

Message::Move { x, y } => {

println!(

"Move in the x direction {} and in the y direction {}",

x, y

);

}

Message::Write(text) => println!("Text message: {}", text),

Message::ChangeColor(r, g, b) => println!(

"Change the color to red {}, green {}, and blue {}",

r, g, b

),

}

}

demonstrates how each value of an enum can be pattern matched.

Nested enums can be destructured similarly to how ChangeColor is matched above.

We can destructure tuples along with structs as well,

let ((feet, inches), AB { x, y }) = ((3, 10), AB { x: 3, y: -10 });

works.

* + - 1. Ignore pattern: We can ignore values by catching all values.

For ex.:

fn foo(\_:i32, y:i32) {

…

}

foo(2,3);

works and foo can only use y. ‘\_’ catches all but doesn’t allow reading the value so it is like an ignoring operator.

‘\_’ can be nested too, match can take tuple values too, combining them both we get

let a= Some(2);

let b=Some(3);

match (a,b) {

(Some(\_),Some(\_)) =>… ,

\_ => … ,

}

says match all values of a,b where a and b are Some values.

Similarly, we can ignore parts of tuples too,

let a= (1,2,3,4,5);

match a {

{first, \_, third, \_, \_ } => …

}

This is an exhaustive pattern but anyway we get 1st and 3rd value only.

\_ can be used to remove unused variable warning, for ex.:

let \_a=2;

even if unused wouldn’t cause a warning.

\_ doesn’t bind to a value by itself but \_<var> does bind to a value. So,

let b= Some(String::from(“yo”))

if let Some(\_a)= b {

…

}

works and \_a takes ownership of String (copy/move behavior).

However,

if let Some(\_) = b {

…

}

also works, but here \_ doesn’t bind and hence the ownership isn’t transferred meaning b can be used later as well, which wouldn’t be the case if we had used any variable inside Some().

* + - 1. Ignoring values with .. : \_ catches all and hence ignores a single value. .. catches all and ignores multiple values.

For ex.:

let a= (1,2,3,4,5);

match a {

{first, .. , last} => …

}

works as if we had given \_ for each value. However, ignored values should be unambiguous, if we had given {.., second, ..} then it would have been an error as it’s a pattern that can match to multiple values.

* + - 1. @ Bindings: Using @ we can bind values inside enums to variables whilst testing them. For ex.:

enum AB {

Hello {x : i32}

}

let a= AB::Hello {x: 1};

match a {

AB::Hello { x: 20 } => println!("{}",x),

AB::Hello { x } => println!("{}",x),

\_ => println!("ya"),

}

arm 1 is an error as x isn’t defined, however arm 2 is not as it catches all values of x and stores it in x.

In cases like these, we have to use @ to indicate variables to be binded and a pattern for condition after that. For ex.:

arm 1 becomes

AB::Hello {x : y @ 20} => …

stores value of x in y when x is 20. The variable name can be same as the field, so x in this case, and it will still work as it shadows the field inside the expression.



1. Memory: Rust is a systems programming language so it allows memory access like C++. All programs have to manage their memory usage, i.e. (de)allocating memory. Other languages either run a GC for the same or leave it upto the programmer (C). But in rust, a third method, ownership is used. Using this method memory (de)allocation rules are checked right at compile time and the program doesn’t even compile if they aren’t followed.
   1. Stack vs Heap: Stack is a fixed size contiguous (adjacent, touching borders) segment of memory. This is a LIFO approach as data must be pushed and popped. In a heap, the size isn’t fixed so when we want to store some data, the chunk is allocated and the address returned through a pointer. Pushing to the stack is faster than to the heap, because the allocator has to allocate the space and do a few other things. Same for reading. When we read from the stack, the entire stack is moved to the processor and then elements can be accessed very fastly. However, since a heap is not a contiguous segment of memory, it can’t be passed all at once to the L caches. So the read is slower. Stack and heap aren’t treated much differently in other languages but in Rust they are, and ownership rules help in understanding how Rust operates with heaps.
   2. Ownership rules:
      1. Each value in rust has an owner.
      2. There can only be 1 owner at a time.
      3. When the owner goes out of scope, the value will be dropped.
   3. Variable scope:

For ex.:

{

let s = “aa”; //immutable string

}

In this example, s isn’t valid before being declared and is only valid till the end of the scope, after which it goes out of scope and is hence disposed.   
  
Variables also go out of scope after their last usage in the code. But this also means, that if a variable is used at the very end of the program then it will remain in scope until then. This ability of a compiler to tell if a variable is no longer is being used before the end of the scope block is called Non-Lexical Lifetimes (NLL).

* 1. Heap: Just like RAII (Resource Acquisition Is Initialization) works in C++, the values on heap in rust are deallocated when the scope finishes.

For ex.:

{

let s= String::from(“yo”) //mutable string

}

Just like previous example, the s loses its value after the scope is finished. But this string is on the heap, so it actually has to deallocate the memory allocated for the strings. For this, rust requires the type to implement drop function. This function is called at the end of the scope and must clear the memory. Here s is the owner of the data “yo” and when it goes out of scope, the value is dropped.

* 1. Copy vs move:

For ex.:

let a=2;

let b=a;

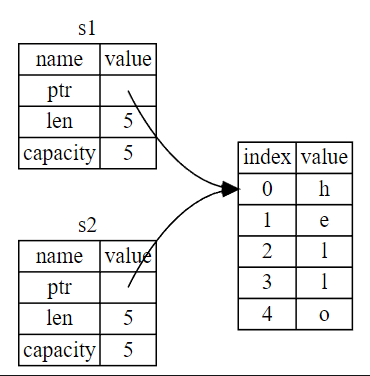
Since this is a simple data type and is on the stack, it is copied directly.

But

let s1 = String.from(“hello”);

let s2 = s1;

here s1 is on the heap and hence the thing that will be copied is the metadata but not the data itself.



and now if we modify the value in the heap then both the variables will be affected.

This would cause a problem in other languages, the double free problem. If s1 and s2 both try to free the memory then it will cause memory corruption. Rust counters this by invalidating the copied variable. Hence,

let s1 = String::from("hello");

let s2 = s1;

println!("{}, world!", s1);

is an error.

This turns the copy operation into a move operation automatically.

Here, s1 is the owner of the data. And when we pass s1 into s2, then s2 takes ownership of the value and in doing so makes s1 invalid as it doesn’t have the ownership anymore.

To actually copy the data, we use the clone method. So s2= s1.clone(); This is an expensive operation as the heap will be recreated.

The same is not true for values on the stack such as integers, they remain valid and copy or clone() are the same thing. As a general rule, all scalar types use copying and not moving. And tuples also use copying if they consist of types that implement copying.

This copy/move scenario works everywhere, even with function calls. So passing a heap variable to a function will turn the variable invalid.

* 1. References: We can use references to pass only the pointer around. This way we neither copy nor move the value itself.

fn main() {

let s1 = String::from("hello");

let p= &s1;

sec(&s1);

}

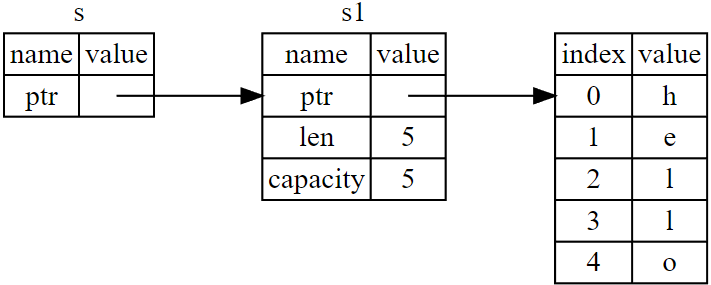
fn sec(s: &String) {

println!("{}",s);

}

Here, we do ‘borrowing’. When we pass &s1 we are borrowing it, which is to say the function sec or p is not going to own the value “hello”. Borrowing is a synonym to ‘referencing the value’ from other languages.

Borrowed values cannot be modified, so sec can’t modify s as it is a reference. This is how the relation looks like



Since s does not have ownership of the value, even if it goes out of scope the value will remain. And even if another variable takes the reference, the value will remain in the primary variable. But if sec had taken a normal value and we had passed s1 directly then s1 would have given the ownership to s and since the function doesn’t return a value, the value would be lost as s1 would’ve passed the ownership to s, and s would’ve gone out of scope after the function.

These immutable references can be used freely.

* + 1. Mutable references: These references allow modification of the underlying data. But they are very strict about their usage.

For ex.:

fn main() {

let mut s = String::from("hello");

sec(&mut s);

}

fn sec(s: &mut String) {

println!("{}",s);

}

Here, sec is allowed to mutate the value in the reference, and the various mut are needed. The first mut defines s as a mutable variable, &mut s borrows a mutable reference and &mut String accepts a mutable string. This works.

However, it has a big restriction. if a value has a mutable reference to it, then it cannot have any other reference at that point of time. So, if we tried

let mut s =…

let p = &mut s;

let x= &mut s;

then this’d be ok, as p loses mutable reference by the time x needs it. However if we then

println!(“{} {}”, p,x);

then this’d give an error as p is still in scope and x is trying to borrow another mutable reference to the same value.

Similarly, when we mix immutable references and mutable references, if an immutable reference is still in scope then borrowing a mutable reference is an error.

For ex.:

fn main() {

let mut s = String::from("hello");

let p = &s;

let x = &mut s;

}

is not an error as p goes out of scope by the time x needs s.

* + 1. Dangling References: The rust compiler uses a concept called ‘lifetimes’ to capture dangling references and prevent them. So,

fn main(){

let p = sec();

}

fn sec() -> &String{

let mut s = String::from("hello");

&s

}

is an error.

As in this function, we are creating a variable on the heap and then returning its reference. But s goes out of scope and since it is still the owner the value is dropped. However, we are returning a reference or borrowing the value and putting it in p, which means p points to an invalid address. This’d be a bug but Rust compiler prevents it through ‘lifetimes’.

* + 1. Slice type: Types like String can be sliced. Slices can be references of parts of string.

For ex.:

let s = String::from(“Hello”);

let x= &s[..]; //entire string

let y= &s[..5]; 0 to 5, end exclusive

let z = &s[2..]; 2 to end

Similarly,

let p = &s[..];

let x = &mut s[..];

both are valid.

String slices are of the type str.

This is also why,

let s =”Hello”;

is immutable, as it is of type &str and is hence an immutable reference or a slice reference. It is called a string literal.

str is specially useful in functions, because not only can they take slices, but also String type.

So,

let p= String::from(“hello”);

let x: &str = &p; //works as &p is coerced into &p[..].

However, while a String literal is a statically borrowed slice, it’s still &str. This means, it is basically a reference to a string in memory however compiler treats it like a static string so we can pass an &str (static string) but it is incompatible with another &str(&str from String, a dynamic string) in another scope as the lifetime isn’t automatically managed. So,

let value = 3;

let yo = match value {

1 => "1",

other => {

let someOtherValue = other.to\_string();

&someOtherValue[..]

}

};

is an error as “1” is a statically borrowed str however &someOtherValue[..] is not, even though they both return &str. This is why, it is advised to return String with “somestring”.to\_string() when dynamic &str are concerned.

Slices aren’t limited to strings. For example they work on arrays too,

let a = [1,2,3,4,5];

let b = &a[..];

Here the type of b is, reference of an array of i32 types, or &[i32].

We can use it as type too, so

fn some(mylist: &[i32]) {

for &item in mylist{

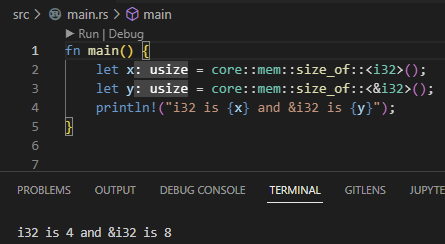
…

}

}

Here, &item takes reference of each item in mylist. Do recall from c++ that the size of a reference/pointer is fixed and it may be larger than a primitive type like int32 so don’t use refs for primitives. They do make things easier though.

As can be seen here



1. Struct: A struct in Rust is used to create custom types and is very similar to a tuple.

Syntax:

struct <name>{

<field\_name>:<field\_type>,

<field\_name>:<field\_type>,

}

For ex.:

struct User{

active: bool,

name: String,

}

and to instantantiate it,

let user= User {

active: true,

name: String::from(“yo”),

};

And to access any value,

let pp= user.active;

But it is an immutable instance, and we cannot modify any field’s value or the variable itself, so to create a mutable instance we use

let mut user …

i.e., Rust doesn’t have mutable fields, the entire instance should either be mutable or immutable.

* 1. Field init Shorthand: Rust allows creating a struct without specifying field names if the variable name is the same.

For ex.:

let active=true;

let user= User{

active

};

* 1. Struct update syntax: A struct can be created from another struct without manually specifying each field,

For ex.:

let active= true;

let user= User{

active,

name:String::from("yo")

};

let user2= User{

..user

};

println!("{}",user.active);

We can specify some fields if we want but we must specify them before defining the base struct.   
Now it copies/moves the values from base struct into the new instance. The copy/move rules are followed the same as described earlier in this doc. This makes the field of the base struct which lost ownership in the move operation as invalid, however all the other fields can be accessed.

So, user.name in println! above would cause an error.

* 1. Tuple struct: We can create a struct which is exactly like a tuple but just with a custom name and is a custom type itself. For ex.:

struct Color(i32, i32, i32);

then to create the instance,

let color= Color(0,1,2);

and access with

color.1; //like normal tuples.

struct Color(i32, i32, i32);

struct AnotherColor(i32, i32, i32);

let color=…

let another\_color=…

color and another\_color are not equal, even if they had the same values and fields as both are different types.

* 1. Unit-Like struct: Just like unit tuples, we have unit structs too.

For ex.:

struct Abc;

let abc= Abc;

and that’s it.

* 1. Reference field types: We can have reference field types, like &str, if we implement lifetimes. This is to ensure the struct owns the data until it is there to avoid bugs.
  2. Methods: Structs can define functions in their own context. However, these methods need to be defined in implementation blocks, which are like cpp files of header files in c++.

For ex.

struct Rect{

length: i32,

breadth: i32

}

impl Rect {

fn area(&self) -> i32 {

self.length \* self.breadth

}

}

And to use it,

let rect= Rect(2,4);

let area=rect.area();

The impl blocks puts the method inside the Rect type. A single struct can have multiple impl blocks defined for it.

Every method of a struct needs to get self as the first parameter in order to access the instance values, it can choose to not have the parameter if it doesn’t need access to field instances, then it is called an associated function. self is of type Self and copies the instance field values. &self is the immutable reference and &mut self is the mutable reference variant. Rust allows us to just specify self as parameter and not specify the type as a shorthand syntax.

* 1. Methods in rust can have the same name as fields.

so length field and fn length() are valid.

Rust know what we are calling by assessing if we specify the () or not.

* 1. Automatic referencing and dereferencing: We know in let x= SomeStruct{} x is an object pointer, and in langs like C, we access them with x->someMethod(). But rust doesn’t need to be specified if we want to use the immutable ref, mutable ref, or copy of the instance as it automatically determines the right reference type.

So,

x.someMethod()

is similar to

(&x).someMethod()

Rust determines the right reference type by looking at the self parameter of someMethod().

i.e., Rust makes borrowing implicit for methods.

* 1. Associated Functions: Methods in the struct which don’t define a self parameter.

For ex.:

struct ABC {…}

impl ABC {

fn someFunc() …

}

then to call someFunc, we use

ABC::someFunc();

someFunc is namespaced by ABC struct so we access it this way.

* 1. Fully Qualified Syntax for Disambiguation: We can define methods with same names as long as they aren’t defined in same scope. Then we call them with their full syntax. For ex.:

trait Pilot {

fn fly(&self);

}

trait Wizard {

fn fly(&self);

}

struct Human;

impl Pilot for Human {

fn fly(&self) {

println!("This is your captain speaking.");

}

}

impl Wizard for Human {

fn fly(&self) {

println!("Up!");

}

}

impl Human {

fn fly(&self) {

println!("\*waving arms furiously\*");

}

}

Here if we say let x= Human{};

then call x.fly() then Human’s fly() will be called. In ambiguities like these, Rust calls the method on the Human type. To call the other methods,

Wizard::fly(&x);

and so on. This is also to say, methods that take self as a parameter can be explicitly given an instance, given the type of instance implements the type for which the method is called.

In case of associated functions, if fly was an associated function (without self) then we would need to cast, so we would call,

Human::fly();

and for others,

<Human as Wizard>::fly();

1. Print and debug: print! is a macro that prints text to stdout. println! is its newline ending variant.

To print stuff

prinln!(“{} yoo {}”,2,4);

where {} indicates scalar/printable data type and “…” is the string slice passed to it.

We can specify output formats inside the {}. If we use {:?} then we specify the debug trait. We can debug print our types with this trait along with an attribute specified on the type,

For ex.:

#[derive(Debug)]

struct …

println!(“{:?}”, somestruct);

and it will debug print it. :#? pretty prints the struct instance.

Alternatively, we can instead use dbg!(<value>). This macro returns the ownership of expressions so having it there or not there is the same thing to Rust.

To use it,

dbg!(somestruct); //and it works,

or

let a=dbg!(2\*4); //a will be 8 and it will print 2 \* 4 = 8

dbg! prints to stderr.

There’s also eprintln!(…)

Just like dbg!, prints stuff to stderr.

1. Enums: Short for enumerations, as they allow us to enumerate the possible variants of a defined type.

To define it,

enum SomeEnum {

A,

B

}

then create their instances them with

let a: SomeEnum = SomeEnum::A;

* 1. Enum types can also be tuples, so

enum A {

X(i32,String),

Y

}

let x= A::X(2, String::from(“yo”));

* 1. Similarly, enums can also have types with named fields.

enum ABC{

A{x: String, y: i32}

}

and to instantiate it,

let a= ABC::A{x: String::from("Yo"), y: 2};

* 1. Enums can also have methods just like structs. They can be declared on the instance (self parameter) or on the type (non-self parameter) and called in a similar way.

For ex.:

enum ABC{

A{x: String, y: i32}

}

impl ABC {

fun yo() -> i32 {

2

}

}

then to use this method,

let a= ABC::yo();

* 1. Option type: Rust doesn’t have the concept of nulls, however we have an enum Option, just like std::optional in c++.

It is pre-defined like so,

enum Option<T> {

None,

Some(T),

}

and it basically means a value either holds a value of type T or None. It is available in the prelude of Rust.

Rust also understands Some and None directly as it shorthands it.

For ex.:

let a= Some(“abc”); //type is Option<&str>

let b: Option<i32>=None;

To get value from an option type, we can use match,

let value =2;

let some= Some(4);

let result = some+ value; // is an error as the type of some is Option<i32>

To add them,

let result = match some {

None => -1,

Some(i) => i+ value,

};

* 1. Enum pattern value binding: Enums in match have their value binded to a given variable automatically.

For ex.:

enum Yo {

A(i32),

B

}

needs a match like,

let x= match a {

B => "na",

Yo::A(value) => "ya"

};

where value is a variable that binds to the value given to A if this arm matches.

1. Packages: Rust, through Cargo, allows multiple files and modules for managing big programs/projects. We can have multiple binary crates but only 1 library crate per package. Cargo also provides workspaces for larger projects. Additionally, scopes are used to set apart parts of program. All these various ways to organize a project’s content is called the module system.
   1. Module System Parts:
      1. Package: One or more crates that provides a set of functionality. Every package needs a cargo.toml file to define how to build the crates. Every package needs at-least 1 crate, and it can contain at-most 1 library crate but can contain any number of binary crates.
      2. Binary Crate: A crate that can be compiled to an executable. It can have a main function.
      3. Library crate: They define functionality for a package, don’t compile to an executable and don’t have a main function. A single package can only have 1 library crate. The main purpose of a library crate is to open access to other crates. Generally we use library crate to define all <use crate> and define all modules. Modules only need to be opened with mod once in a crate, and only library crate can provide access to the root crate from external crates. This is why it is the perfect place to open modules and permit Integration Tests, without it Integration Tests can’t access other crates not even the root crate and hence can’t work.
      4. Crate root: A source file that Rust compiler starts from and then it compiles the root module of our package.
   2. Folder structure: When we create a new project with cargo new, cargo creates a folder with package-name with a cargo.toml file in it and places a folder named src in it as well. Inside it, a file called main.rs is created. This file is the crate root of the binary crate and this binary crate is named the same as the package. We can create a src/lib.rs and this file is assumed as library crate of the package, the name of this crate is the same as the package as well. If a crate contains both main.rs and lib.rs then 2 crates are created for the package, with the same names as the package. All the other binary crates can be placed in the src/bin directory, every file inside it is treated as a separate binary crate.
   3. Modules: A module is basically a container for 0 or more items, i.e. structs/funcs/etc. This is how we declare, use and work with modules,

For ex.:

//On top of an rs file

mod x {…}

First the crate root is opened, i.e. either main.rs and if it’s not there then lib.rs. Then it goes in them and looks for mod xyz; lines at top. These lines define which module to open. Then to look for the code of the module,   
the compiler first checks if the line is followed by {…} instead of a semicolon, if it is then that’s assumed as the code for it. If not found then it goes to src/modulename.rs. If not found then it goes to src/modulename/mod.rs.

* + 1. Submodule: If a module declares its own modules then it follows a similar approach,

first it checks inline,

then it checks src/modulename/submodulename.rs

then it checks src/modulename/submodulename/mod.rs

and similarly for its submodules.

* + 1. When a module is compiled for a crate, the module can then be accessed by any other module in the same crate. To do so,

we access

crate::modulename::submodulename::SomeMethod

given the privacy rules allow.

* + 1. Private vs Public accessor: A parent module can’t access the submodules used by its children if they don’t declare them publicly. By default, all items in Rust are private and a parent module can’t reference it’s children if they are private, the reverse is allowed, as the child needs to know its parent. To make items public in rust,

mod -> pub mod

fn -> pub fn

and so on.

Since all items private by default, they need to be made publicly accessible as well. So we give pub to items to make them public.

For ex.:

pub struct …

i.e., even if we make a module public, it’s items still remain private until explicitly made public.

* + - 1. Struct fields and methods are private unless explicitly made public with pub. For enums, we just need to make the enum public to make all its fields and methods public.
    1. Path shortcut: Instead of a crate::modulename::submodulename::SomeMethod each time we need to access the method, we can use the ‘use’ keyword. It’s like using namespace from C++ and defines the item for the scope. So,

pub mod modulename; //Firstly declaring the module

use crate::modulename::submodulename::SomeMethod;

//now we can directly use SomeMethod in the rest of the program //as SomeMethod is a valid item in the scope.

‘use’ simply creates a shortcut for the absolute/relative path of an item in the scope it is declared in (and it’s children). However, modules have their own scopes, so

mod x{

pub fn y() {}

}

use crate::x::y;

mod z{

pub fn m(){

y();

}

}

y() is an error as mod z has its own scope. However, since they are sibling modules, we can use

super::y();

and this will work. As super looks for y in the scope of the parent, and in that scope y is a valid item because of the use.

* + - 1. It is advised to not use ‘use’ and open the item itself in the scope, but its parent. This is a more Rustacean way and helps distinguish local items from separate packages as well as avoid naming conflicts if the same item exists in other packages/local scopes.

For ex.:

mod x{

pub fn y()

}

use crate::x; //instead of crate::x::y;

then use it with

x::y(); //instead of plain y();

* + - 1. as: We can use ‘as’ with ‘use’ to give different name to an item.

For ex.:

use crate::x as z;

z::y();

or

use crate::x::y as z;

z()

* + - 1. Re-Exporting: Using ‘pub use’ we can re-export an item opened with use. This allows others to use the opened item from outside the scope.

For ex.:

pub mod modA {

pub fn fnA() {}

}

pub mod modB {

pub use crate::modA;

pub fn fnB() {

modA::fnA();

}

}

mod modC {

use crate::modB;

pub fn fnC() {

modB::modA::fnA();

}

}

In the above example, modB opens the modA in its scope and re-exports it. Meaning when modC opens modB then it can access modA through modB as modA is a valid item for modB and modC knows that as well.   
However, if we remove pub from pub use crate::modA; in modB then it won’t re-export it and modC wouldn’t see modA as a valid item in modB’s scope.

* + - 1. Nested Paths: Using ‘use’ for each item in an item is too tedious, so we can use a single ‘use’ to open multiple items.

For ex.:

use crate::x::{y, z, m::n};

We can also nest the base path if we wish to open it too,

use crate::x::{self, m::n};

opens crate::x; and crate::m::n;

* + - 1. Glob operator: Used to open all items in an item.

use crate::x::\*;

opens all items in x in scope. Generally used in testing as in production projects this pollutes the scope.

* + 1. ‘crate’: The reason lib.rs and main.rs are given the same package name is because they both make up the crate root and the ‘crate’ module. This is the first module that’s created and all submodule crates are inside it.

The root module is called ‘crate’ module. This is why when we use ‘use’ we start from ‘crate’ as ‘crate’ is rootmost module, which has the content of main.rs and lib.rs.

For ex.:

For a code:

mod front\_of\_house {

mod hosting {

fn add\_to\_waitlist() {}

fn seat\_at\_table() {}

}

mod serving {

fn take\_order() {}

fn serve\_order() {}

fn take\_payment() {}

}

}

fn main() {…}

The module hierarchy looks like this,

crate

└── front\_of\_house

├── hosting

│ ├── add\_to\_waitlist

│ └── seat\_at\_table

└── serving

├── take\_order

├── serve\_order

└── take\_payment

* + 1. Path to item in a module:

2 ways mainly,

Absolute Path (crate::something…)

Relative Path (self, super or an identifier in current module)

For ex.:

mod front\_of\_house {

mod hosting {

fn add\_to\_waitlist() {}

}

}

pub fn eat\_at\_restaurant() {

// Absolute path

crate::front\_of\_house::hosting::add\_to\_waitlist();

// Relative path

front\_of\_house::hosting::add\_to\_waitlist();

}

* + - 1. Relative path:

We can access items in the parent module with super. For ex.:

mod yo{

someMethod();

mod pp {

fn bb() {

super::SomeMethod()

}

}

}

* + 1. Best Practises:
       1. We should define modules in lib.rs instead of main.rs. And define only the necessary amount in the main.rs to kick-off lib.rs. This is because the code in the lib.rs can be shared with other packages, which allows a better design for a reusable package as everyone can directly see lib.rs and modify their own packages as necessary to use lib.rs. This way, our main.rs itself becomes a client and can only access the public items, which gives us the same behavior as if an external package would.
    2. Separate module per file: We can separate a module into different file. To do so,

For ex.:

For a module,

mod modA {

pub modB{

pub fnA(){}

}

}

In main.rs or lib.rs

First declare the module,

mod modA;

//Now Cargo will look for modA.rs in src.

Create modA.rs in src with content

pub mod modB;

Then compiler will look for modB in a folder named src/modA.

Create modB.rs in src/modA (folder) with content

pub fnA() {}

and that’s it.

Alternatively, an older method of folder structure can be followed.

src/modA/mod.rs //for modA

src/modA/modB/mod.rs //for modB

and so on.

While this seems more intuitive, the same filenames make it confusing for working but to the compiler it makes no difference and we can even mix both folder structures. However each module must only be declared using any one method not both.

* + 1. Using lib.rs in main.rs: It is advised to use lib.rs to declare all crates, open all paths and then define all modules. Then we can simply reference it from the main.rs.

For ex.:

For a lib.rs,

pub fn yo() {…}

the main.rs would be,

use myCrate;

fn main() {

myCrate::yo();

}

and the cargo.toml would be,

[package]

name = "myCrate"

version = "0.1.0"

edition = "2021"

All 3 steps are important. The lib.rs becomes the crate with the same name as the pkg, hence public/private rules are followed even through crate root which is main.rs. We have abstracted away the crate root’s content into a different crate hence even it must access the lib.rs like external crates would. And main.rs needs to open the path to it, the lib.rs items aren’t in main.rs’ scope.

Lastly, the cargo.toml definition describes important metadata. Even without “edition” key, the main.rs wouldn’t be able to access the lib.rs.

* + 1. s
  1. Workspaces: A set of packages that share the same cargo.lock and output directory. To create a workspace,

create a folder, say “add”,

then create a cargo.toml inside it with,

[workspaces]

members= [

“adder”,

]

then inside it create the adder package with cargo new adder,

And now adder is a part of add workspace. If we run cargo build it will create adder’s build in add/target instead of add/adder/target. This is to allow crates to depend on each other.

Similarly, add another member

members = [

“adder”,

“add\_lib”

]

then, create a library in add,

cargo new add\_lib –lib

and define a method in add/add\_lib/src/lib.rs

pub fn yo() {…}

Now to use this crate in add,

define so in adder’s cargo.toml,

[dependencies]

add\_one = { path = "../add\_one" }

Then to use the method,

In add,

use add\_one;

…

add\_one::yo();

* + 1. cargo run on a workspace doesn’t work. We need to define a binary package and run it instead, so here we can

cargo run --p adder

runs adder/src/main.rs

* + 1. External packages: When one package uses a version of an external package, others use the same. The version to use is selected by cargo automatically and the cargo.lock generated for the workspace holds that version. Any crate in the workspace will then refer to that version only, although it has to declare the dependency on a package in its cargo.toml.
    2. Tests: cargo test tets all crates in a workspace. We can specify a specific crate with cargo test --p myCrate.

1. Errors: Rust divides errors into 2 categories, recoverable and non-recoverable errors. The former are as Result<T,E> type while the latter are triggered through panic! macro. The reason we have 2 categories of errors is because we may want a program to crash such as when wrong array index is accessed while we may be able to work around a simple error like file not found.
   1. Unwinding vs Aborting: When the program panics, Rust unwinds the stack and walks back to each of the function to cleanup its resources. This is a tedious process which takes time and while enabled also increases size of the Rust executable. We can choose to simply abort on panic, in which case it leaves it upto the OS to clear the resources resulting in a quick exit and smaller binary size.

To do so,

In cargo.toml,

[profile.release]

panic = 'abort'

here we are setting abort on panic for release build.

* 1. Panic: To panic,

panic!(“yo”); //anywhere.

* + 1. Backtrace: When panic is called, it shows the line and character of the line and char of file at the bottom of the call stack. So, for ex.:

fn main(){

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

v[4];

}

This will cause a panic but in some other file where vector is defined. However, Rust will show the line and char of this file since this is the origin of action that lead to the crash.

We can see the backstack using the env var RUST\_BACKTRACE,

like RUST\_BACKTRACE=1 cargo run.

* + 1. Result: When we want to define a recoverable error.

Result is defined like so internally,

enum Result<T, E> {

Ok(T),

Err(E),

}

And it is used like so,

use std::fs::File;

use std::io::ErrorKind;

fn main() {

let f = File::open("hello.txt");

let f = match f {

Ok(file) => file,

Err(error) => match error.kind() {

ErrorKind::NotFound => match File::create("hello.txt") {

Ok(fc) => fc,

Err(e) => panic!("Problem creating the file: {:?}", e),

},

other\_error => {

panic!("Problem opening the file: {:?}", other\_error)

}

},

};

}

or with closures,

use std::fs::File;

use std::io::ErrorKind;

fn main() {

let f = File::open("hello.txt").unwrap\_or\_else(|error| {

if error.kind() == ErrorKind::NotFound {

File::create("hello.txt").unwrap\_or\_else(|error| {

panic!("Problem creating the file: {:?}", error);

})

} else {

panic!("Problem opening the file: {:?}", error);

}

});

}

* + - 1. Turning Result to panic: We can define a variable that only takes the needed value of the Result and panics otherwise.

To do so, we use either unwrap or expect method of File::open.

let f = File::open("hello.txt").expect("Failed to open hello.txt");

f will always be of type File as otherwise it will panic. unwrap() is like expect except it doesn’t take a custom panic string.

Similarly we can define this behavior for our types too.

* + - 1. Propagating errors: We can simply return an Ok or Error from a Result to propagate the error.

For ex.:

…

return match SomeResult {

Ok(s) => Ok(s),

Err(e) => Err(e),

}

There’s a shorter syntax, ‘?’

For ex.:

…

SomeResult?;

return Ok(“ya”);

If SomeResult is an error, it returns it. Otherwise the flow it continues and it sees Ok so it returns that turning the return type of this function Result<String, someerror>

The ? syntax tries to turn the returning error to the type that the function accepts, for ex.:

fn somefunc()-> Result<String,A> {

SomeResult?;

Ok(“yo”)

}

If SomeResult returns a type of error B, then it will go through the From trait in the standard lib, all it needs is a

impl From<A> for B

function in the trait’s from function.   
That is, it tries to coerces the type.

* + - 1. Option<T> and ?: When ? is used on an Option return type, then None is returned if it returns None otherwise Some.

For ex.:

fn last\_char\_of\_first\_line(text: &str) -> Option<char> {

text.lines().next()?.chars().last()

}

If next() returns a None then None is returned otherwise Some is returned and the expression continues and puts Some in chars().

* + - 1. Main’s return type: Main returns () but it can also return a Result. For ex.:

use std::error::Error;

use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {

let f = File::open("hello.txt")?;

Ok(())

}

Box<dyn Error> is the type of error Main accepts and is a trait object, it basically means “any kind of error”. The Main with this return type returns a 0 if Ok and another int otherwise.

* + - 1. When to panic or Result: There’s a general rule when the code should crash and when it shouldn’t. It is defined like so,

It’s advisable to have your code panic when it’s possible that your code could end up in a bad state. In this context, a bad state is when some assumption, guarantee, contract, or invariant has been broken, such as when invalid values, contradictory values, or missing values are passed to your code—plus one or more of the following:

The bad state is something that is unexpected, as opposed to something that will likely happen occasionally, like a user entering data in the wrong format.

Your code after this point needs to rely on not being in this bad state, rather than checking for the problem at every step.

There’s not a good way to encode this information in the types you use.

1. Generic Types: Generics are types we use as abstract stand-ins for concrete types which would be defined elsewhere allowing a method or anything that needs a type to cover a range of concrete types.
   1. Generic:

To define a generic,

fn myMethod<T> Yo(list: &[T]) -> i32 {

for &item in list {

if item > 2 {…}

}

}

while a correct example of generics, fails to compile. This is because the type T isn’t guaranteed to implement the ‘>’ operation, i.e., not all possible types of T would implement >. To allow it to work, we will need to require a trait std::cmp::PartialOrd for type T. We can do that by

fn myMethod<T: PartialOrd> Yo(list: &[T]) -> i32 {

for &item in list {

if item > 2 {…}

}

}

Now it will only accept T which implements the given trait, meaning the code below can rest assured that ‘>’ will be defined on the type.   
Just like in C++ we don’t need to explicitly define the type for T at call-site as it is inferred automatically.

In structs,

struct X<T> {

x: T,

y: T

}

T can only be of 1 type,

So,

let z = X{ x: 4, y: 4.0};

is an error.

For multiple generics,

struct X<T,Y> {…}

Similarly, for enum,

enum Option<T> {

Some(T),

None

}

says Some will have a parameter of type T. This is basically how Option<T> is defined.

For methods,

struct X<T> {…}

impl<T> X<T> {

…

}

Here we say impl will accept a generic and the same generic will be used for an instance of the X. impl can only take a generic if it is applied to the struct/enum. And the type after impl can take a generic only if impl takes a generic.

So,

impl X<i32> {…}

is valid.

impl<T> X<T> {…}

is valid

but

impl X<T> {…}

is not

When we define a method for X<T> then it is on every instance of every type of X. But when we define it for X<i32>, then the method is only on every instance of X of type i32.

The method itself can have its own generics too,

struct X<T> {

yo:T,

}

impl<T> X<T> {

fn y<K>(a: K, b: T)-> T {

return b

}

}

This method says that all instances of X of every type will have a method y that accepts a value of type K and type T and returns value of type T.

* 1. Runtime Cost: 0. This is due to a process called monomorphization. During compilation, Rust compiler creates the definitions for all types of a generic that are used in the code. This makes it as if we had declared each function for a type separately. And the generic is replaced with concrete types at compile time itself. The resultant code is called Static Dispatch, as the compiler knows what to call at compile time itself.
  2. Trait: A trait in Rust is like an interface in other languages, although quite different. Using traits we can define functionality for a given type and also functionality that it can share with other types. There’s also trait bounds which limit the application of a trait to only types that have a certain behavior/functionality already present.
     1. Trait Definition: Just like an interface, a trait defines behaviors which are defined through method signatures. For ex.:

pub trait Something{

fn Moo(&self)-> String;

}

We define this as a public trait, and this implies all method signatures in it are public too. Now to use it,

struct X {…}

impl Something for X {

fn Moo(&self)-> String {

…

}

}

and this will work.

* + 1. Orphan Rule: This is a part of coherence property. According to it, a trait can be implemented for a type iff either the trait or the type is local to the crate. For ex.:

If crate A defines a trait X then it can define the trait X for any type, local or external, such as a type T from crate B.

Similarly, if crate A defines a type T then it can implement trait X defined in crate B for type T.

But, if crate C defines neither type T or trait X but takes them from crate A and B, i.e., both are external then this is an error.

This is to ensure the Parent type exists, and other people’s code can’t break our own or theirs because a trait/type is defined at a place different from the crate where it is created.

* + - 1. newtype pattern: It is possible to avoid the orphan rule by basically creating our own local type, holding a value of the external type and then implementing an external trait on it. Newtype refers to the same name type present in Haskell.

For ex.:

use std::fmt;

struct Wrapper{

v: Vec<String>

}

impl fmt::Display for Wrapper {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "[{}]", self.v.join(", "))

}

}

fn main() {

let w = Wrapper{v:vec![String::from("hello"), String::from("world")]};

println!("w = {}", w);

}

We could have used tuple struct here for a simpler syntax.

* + 1. Default implementation: We can give a default implementation of a method as well, it still needs to be mentioned in the implementing types though.

For ex.:

pub trait X {

fn yo()->String {

return “na”.to\_string();

}

}

struct A{…}

impl X for A {

fn yo()->String{ }

}

Yes, we need to actually leave the block empty.

We can’t call the default implementation from the overriding implementation.

* + 1. self: We use self in a trait to pass an instance of the type to the function, allowing us to call other methods and fields.

While this means we can’t access a type-specific method/field in a trait’s default method, we can access trait-specific methods because of it.

For ex.:

pub trait X {

fn na();

fn yo(&self)->String {

self.na();

return “na”.to\_string();

}

}

works because self will always have na() as it’s defined on trait itself.

* + 1. Traits as parameters: Traits can be used as parameters too.

For ex.:

fn yo(list: &impl X) {…}

says ‘list’ will be of a reference of a type that implements a trait X. Using this approach, ‘list’ can access methods defined for the trait X in this function.

However, it is a syntax sugar for a bit longer syntax,

fn yo<T: X>(list: &T) {…}

says T will be a type that implements a trait X. This is also known as Trait Bound syntax.  
Even though this is the full syntax of it, it is a slight bit different from the sugar.

For ex.:

fn yo(list1: &impl X, list2: &impl X) {…}

the impl trait syntax here says it needs 2 values of types that implement X, but it doesn’t require the type to be the same. And if that is a requirement, trait bound guarantees it as we only use 1 generic type. So when we call the trait bound function we have to pass the same type values as generic takes only 1 type but in the impl trait function the types can be different.

To specify multiple traits we use the ‘+’ operator,

fn yo(list: &(impl X + Y)) {…}

or

fn yo<T: X+Y>(list: &T){…}

* + - 1. Sized Trait: This trait is automatically applied to all types whose size is known at compile time. Since this is a trait, it is present in generics too. And hence can be modified too,

For ex.:

fn yo<T> (a: T) {}

is actually

fn yo <T:Sized> (a:T) {…}

So,

fn yo<T: ?Sized> (a: &T) {…}

allows DSTs to be passed to yo as well. The ?Trait syntax says T may or may not implement the Sized trait. This syntax is only applicable for Sized trait.

* + 1. Where clause: Specifying traits in generics can be a bit hard to read, to assist with that we have where clause in Rust which makes it a bit more readable. For ex.:

fn yo<T: X, R: X+Y>(…) -> String {…}

can be written as

fn yo<T,R>(…)-> String

where T: X,

R: X+Y

{…}

* + 1. Return types with traits: Functions return type can be generic, but it can also be a generic that has traits. For ex.:

fn yo()-> impl X {…}

says yo() will return a type that will implement trait X.

There is a limitation because of how impl trait is defined in the compiler, which is that the return types must be the same.

So,

fn yo()-> impl X {

if true{

return <T type value>;

}

else {

return <R type value>;

}

}

is an error even though both T and R implement X. We have to use trait objects to work around this limitation.

* + 1. Conditionally implementing traits and methods: Using trait bound syntax we can define methods or traits for types that have given traits. For ex.:

struct X<T> {…}

impl<T: Y> X<T> {

fn yo() {…}

}

defines a method yo() for all instances of X that are instantiated with types that have a trait Y.

Similarly,

impl<T: Y> Z for X<T> {

…

}

implements trait Z for X for all instances of X that are instantiated with a type that implements the trait Y. Or in other words, all instances of X with a type T (where T implements trait Y) would implement the trait Z. This is known as blanket implementation as it blankets a lot of types.

* + 1. Derivable Traits: By using #[derive] attribute we can make our types automatically implement a given trait. This only works for a few basic traits like PartialEq. For ex.:

#[derive(PartialEq, Debug)]

struct X{…}

* + 1. Associated Types: They are kind of like generics as they put a placeholder type in the trait itself which the trait can use and then the implementor defines the type and implements the trait.

However, unlike generics, associated types provided a bit of brevity. For ex.:

pub trait ITer<T> {

fn next(&self) -> Option<T>;

}

struct Counter{

x: i32,

}

impl<T> ITer<T> for Counter {

fn next(&self) -> Option<T> {

return None;

}

}

fn main() {

let ctr= Counter{x:2};

ctr.next() as Option<i32>;

}

works but we need to specify the type of next() function. This is because there will be multiple implementations of ITer for Counter like ITer<String>, ITer<i32> and so on, and the next() method needs to know which one to call so it needs explicit type annotation.

However, with associated type,

pub trait ITer {

type Yo;

fn next(&self) -> Option<Self::Yo>;

}

struct Counter{

x: i32,

}

impl ITer for Counter {

type Yo = i32;

fn next(&self) -> Option<Self::Yo> {

return None;

}

}

fn main() {

let ctr= Counter{x:2};

ctr.next();

}

that ambiguity is extinguished. Here, the trait says it needs a type whose placeholder is Yo and the next method in it returns an Option of type Yo (Self::Yo as Yo is a type inside Self) and in the implementation we define that type to be i32. Now next() method call knows which method to call as there’s only 1 of it. Unlike generics, this implementation is the only implementation for ITer for Counter.

* + 1. Supertrait: A trait that gets implemented by another trait. So like interfaces extending interfaces in other languages.

For ex.:

trait X {

fn yo(&self);

}

trait Y: X {

fn pp(&self){

self.yo();

}

}

struct A {}

impl Y for A {}

impl X for A {

fn yo(&self) {}

}

Basically, if A wants to implement trait Y it needs to implement trait X. But the added advantage is that trait Y can directly use methods of X Then, as impl provides definition for all methods in traits, if we call A{…}::pp(); then it works.

* 1. Lifetimes: They are a kind of generic and define the scope of references. By defining the scope of a reference, we define how long it is valid. Just like the type is inferred for generics, the lifetimes are inferred as well. This is a feature that only Rust has.
     1. The Borrow Checker: Rust compiler has a borrow checker to compare the scopes to determine if all the borrows are valid.

For ex.:

{

let r; // ---------+-- 'a

// |

{ // |

let x = 5; // -+-- 'b |

r = &x; // | |

} // -+ |

// |

println!("r: {}", r); // |

} // ---------+

Here we show r has a lifetime of ‘a while x has a lifetime of ‘b. As we can see, b has a smaller lifetime but r is borrowing the value of x and since the owner dies before the borrower, this is an error.

{

let x = 5; // ----------+-- 'b

// |

let r = &x; // --+-- 'a |

// | |

println!("r: {}", r); // | |

// --+ |

} // ----------+

This works as r the borrower dies before x the owner. This code also shows that Rust deallocates r after its last usage.

However, it gets a bit complex when functions are involved, so

fn yo(x: &str, y: &str) -> &str {

if true {

x

}

else

{

y

}

}

It is an error as the Rust compiler can’t determine at compile time if x or y will return so it doesn’t know whose lifetime is to be inspected. Even if we had returned a &String[..] (&str) which was generated inside yo() it’d be an error as it would have a smaller lifetime than any caller.

* + 1. Explicit lifetime definition: We can pass lifetimes around, but we can’t change them. To explicitly define lifetime for a reference,

‘<letter>

For ex.:

x: &’a mut i32;

says x will be a mutable reference of type i32 with a lifetime of ‘a.

For ex.:

fn yo<’a>(x: &‘a str, y: &‘a str) -> &‘a str {

if true {

x

}

else {

y

}

}

works. Lifetime definitions on function parameters are called input lifetimes while the return type lifetime definitions are called output lifetimes.

This func says x and y will take the same lifetime parameters and the function will return a value with the same lifetime.

Here we are not changing the lifetimes of x, y or return type. Rather, we are simply only allowing values with the lifetimes as defined in the signature. In the above example, while without the lifetime generic the code failed to compile, it was only compiling after it because it would only run when the given lifetimes are provided, otherwise it’d crash.

So,

if we call yo

let a= String::from(“ye”);

let result;

{

let b= String…

result= yo(a.as\_string(),b.as\_string());

}

println(“{}”, result);

It will be an error.

This is also because when multiple lifetimes can be inferred, the shortest lifetime is passed to the lifetime generic.

The lifetime generic can only be applied to a function signature and not it’s body. Meaning if we declare a variable in the body we can’t use the passed lifetime generic for it.

* + 1. Lifetimes with structs: For ex.:

struct X<’a> {

yo: &’a str,

}

says the instance of struct X will only live as long as the string yo lives.

* + 1. Lifetime with impl:

impl<’a> X<’a> {

fn yo(&self) {…}

}

* + 1. Lifetime generics are always required when returning references. However, some syntaxes are already defined for the compiler to ease up boilerplate.

For ex.:

fn yo(x: &str) -> &str {

…

&x[0..2]

}

works even though it shouldn’t. This is because Rust compiler implicitly adds <’a> to the signature. These pre-defined rules are called lifetime elision rules and aren’t of much importance to us but to the compiler and when a use of reference fits a rule then we don’t need to specify lifetimes.

* + 1. Lifetime elision rules: Functions and impl blocks always need a lifetime definition for reference parameters & reference return types. But if we don’t provide them then these rules determine if it’ll be an error or not. There are 3 of these:
       1. Each reference parameter in signature gets a separate lifetime.

fn yo(x: &str) gets fn yo<’a>(x:&’a str), fn yo(x…,y…) gets fn yo<’a, ‘b>(x: &’a str, y: &’b str) and so on.

* + - 1. If there’s exactly 1 reference parameter and it is given an explicit lifetime then the same lifetime is applied to the return type reference, if any.
      2. If there are multiple input lifetime parameters, and one of them is &self or &mut self then the self’s lifetime is applied to the return type reference.

If any of the 3 rules can’t be applied to a method/function and it doesn’t have it’s own explicit lifetime definitions for the reference types then it is an error. Multiple rules will be applied to the same signature if a single rule doesn’t make the signature valid.

For ex.:

impl … {

fn yo(&self, x: &str) -> &str {…}

}

This is not an error, as the 1st rule applies a lifetime to both the parameters and the 3rd rule applies the self’s lifetime to the return type.

* + 1. Static lifetime: Applying this lifetime to a reference puts it in the binary and increases the size of it. But the reference is valid for the duration of entire program.

For ex.:

let x: &’static str= “yeee”;

works. All String literals have the static lifetime.

* 1. Default generic type parameters: We can define default type for generics if they aren’t given with the type annotations. They are only applicable for generics on structs, enums, types or traits. For ex.:

struct X<T=i32> {

x:T,

}

let yo = X{x:2};

Although this isn’t an ideal use case as x’s type is being passed as i32 anyways.

However, if we look at Add trait in std::ops

trait Add<Rhs=Self> {

type Output;

fn add(self, other: Rhs) -> Self::Output;

}

then it becomes apparent that even if impl Add on X is called it will still work as Rhs will default to X in that case.

* 1. Explicit Type Annotation: To explicitly give type for the type annotation, we use :: with <T>. Also called the ::<> or turbofish operator, (<https://turbo.fish/>).

For ex.:

fn yo<T>(x:T){}

let x= yo::<i32>(2);

1. File I/O:
   1. Read file to string:

use std::fs;

use std::io;

fn read\_username\_from\_file() -> Result<String, io::Error> {

fs::read\_to\_string("hello.txt")

}

1. Testing: All tests are written in 3 steps, create the proper state, run the function with the proper state and lastly check the result. Tests in rust are functions annotated with #[test] and a test module is generated automatically for any new cargo library created with cargo new <name> --lib.

To run tests, we run   
cargo test

* 1. Basic test:

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

let result = 2 + 2;

assert\_eq!(result, 4);

}

}

The attribute on module defines this module as the test module.

Any method without this attribute isn’t ran with the test runner.

It will give the following strings in its output when ran with cargo test.

1 passed  
0 failed  
0 ignored //we can ignore tests

0 measured //for benchmark tests, currently only on nightly Rust

0 filtered out //we can specify only certain tests to run using cargo test

* 1. Failure: Tests can fail if asserts fail or if a test method thread dies. Each test is ran in a different thread and if a thread becomes unresponsive or dies it is marked as failed. For ex.: test methods that panic!.
  2. Accessing main crate: The test module in library (src/lib.rs) is created in the crate root but inside a test module. So to access the other parts of crate we use

use super::\*;

This opens the scope of the rest of the crate root to the test module.

Asserts: The available check macros.   
For ex.:

* 1. assert!(false); // Checks a condition, and panics! if it results in a false.

assert\_eq!(1,2); //is 1 equal to 2

assert\_ne!(1,1) //is 1 not equal to 1

* + 1. Custom failure text: After the required arguments, the string passed is given to format! macro and it prints given string.
    2. should\_panic: Using this attribute we can check if a test method panics.

#[test]

#[should\_panic(expected = "Yeee")]

fn yo() {…}

now if the method doesn’t panic then it is a failure. On panic the given optional text is printed and is success.

* + 1. Result: Test methods can return Result instead of asserts or panics to check for failures.

#[test]

fn yo() -> Result<(), String> {

if true{

Ok();

} else {

Err(“lol”);

}

}

works.

As for Result shorthand with ? which means panic on Err, we can use

let value= SomeResult;

assert!(value.is\_err());

This is a bit faster than letting the method panic and die with ?.

* 1. ignore: We can ignore certain test methods by using this attribute.

For ex.:

#[test]

#[ignore]

fn yo(){…}

this test method will be ignored.

* 1. Cargo test CLI options: Normally all tests are run parallely using cargo test. We can configure it and other options using the CLI args. However, CLI args are passed to cargo test for both the test runner and the test binary and hence must be separated. They are separated by --.

For ex.:

cargo test --help

is passed to the test runner, displays test options.

cargo test -- --help

is passed to the test binary, displays test binary options.

* + 1. Test binary options:

cargo test -- --<option>

* + - 1. test-threads=1 :To run tests on 1 thread only, i.e., consecutively instead of parallely.
      2. show-output :To show println! or format! macro outputs in the test results.
      3. ignored :Runs only the ignore attributed tests.
      4. include-ignored :Runs all tests including ignored ones.
    1. Test runner options:

cargo test --<option>

* + - 1. tests <filter>: To run given name tests
      2. test <filename>: Runs the given IT file only.
    1. Test args: These are passed directly after the cargo test and act as shorthand syntaxes for test runner options

cargo test <arg>

* + - 1. <filter>: By just defining a name in the filter, only tests with the given value are run while others are filtered out.

This is the same as

cargo test --tests test::<filter>

* 1. Basic procedure: Tests are divided into 2 cats, unit tests (to test only methods or values) and integration tests (to test a functionality or a feature like class is processing some data correctly, generally includes multiple methods). UT are the ones above, IT is defined later. By convention, each rs file should define its test module at the top, these are the UT tests. The name of test module doesn’t matter but it’s generally tests. Similarly, the cfg attribute isn’t necessary either.

#[cfg(test)]

mod …

The cfg attribute tells the Rust compiler to not include the module in normal builds but only in cargo test builds, this saves compile time. The cfg stands for configuration, and tells Rust compiler to only include the given item in binary on given option, here it is test.

* 1. Private functions: Test methods can freely access private methods.
  2. Integration Tests: Tests out functionalities. They are created in their own crates (outside crate root) and need to be in tests/<somefilename>.rs where ‘tests’ is a sibling folder of src.

For ex.:

In myCrate/tests/myTest.rs

use myCrate;

#[test]

fn yo() {…} //normal test method

This test is only compiled when cargo test is ran.

* + 1. Submodules: If we place any rs file in the IT dir then the file is taken by the test runner and then ran., even if it doesn’t have a #[test] method. To prevent this we place these modules in any sub-directory.

For ex.:

fn yo(){…}

in tests/commonss/mod.rs

Now to use it,

use myCrate;

mod commonss;

#[test]

fn ya() {

commonss::yo();

}

The files in subdirectories aren’t picked up by the test runner.

* + 1. IT can’t work with root crate or any other crates if the package doesn’t have a library crate. This is because the binary crate is directly compiled to an executable and hence can’t be used by external crates. We define main functionality in the library crate and even open modules in it. This allows external crates, including IT crates to access it (the crate root) and all other modules as well.

1. Functional Programming: Rust borrows some concepts from FP languages. An FP language generally sees functions as values.
   1. Closures: Anonymous functions. Can be stored, passed around, capture the values of their context and even be ran in different contexts.

For Ex.:

let someOption= Option::Some("ya");

let value=someOption.unwrap\_or\_else(|| someFunction());

Here someFunction is passed to unwrap\_or\_else if someOption has None. someFunction is called a closure in this case as it is passed to the method, the method here needs a function which takes 0 arguments (&self isn’t included) and returns a value of type T where T is the type of Some. So, if someOption was None then it would run the function, get an &str value from it and then pass it to ‘value’.

* + 1. To define a closure,

let myClosure = |yo: i32| -> i32 {

…

somei32

};

Unlike functions, closures generally don’t need to explicitly define arg types or return types as that can be inferred automatically because Closures are short-lived, context-dependent and narrow-use functions.

So,

let myClosure= |yo| {

…

somei32

};

or

let myClosure= |yo| 2\*yo;

are valid too. But since their types aren’t defined right away, they need to be defined later in the program otherwise it is an error.

myClosure(2); to call it.

The last syntax without any params looks like,

let myClosure= || println!(“yuh”);

* + 1. Capturing variables: Just like the parameters have mutability rules like normal functions, the captured variables too.

To capture a variable,

let list= vec![1,2,3];

let myClosure= || println(“{}”, list[0]);

captures list immutably.

let myClosure= || list.push(4);

captures list mutably. We know mutable references don’t allow immutable references until their scope ends. So,

let list= …;

print!(…,list[0]);

let myClosure= || list.push(4);

myClosure();

works, but

let list= …;

print!(…,list[0]);

let myClosure= || list.push(4);

print!(…,list[0]);

myClosure();

doesn’t work. According to the borrow checker, the closure maintains a mutable reference until the last call to it. Just like normal variables.

Closures capture used variables in their context automatically, so even if we pass it to this

fn yo<T>(a: T ) where T: Fn() {

a();

}

yo(|| myClosure());

It will work.

* + 1. Closure Traits: Closures automatically implement traits based on how they deal with the captured references.

The traits are applied additively, so more than 1 can be applied to a Closure, they are:

FnOnce: Applies to all closures that can be called at-least once. All closures implement this. Ones that move captured references outside their body implement only this because they can be called only once as they move the references outside so they can’t be used again.

FnMut: Applies to closures that mutate references but doesn’t move them outside.

Fn: Applies to closures that don’t mutate references and don’t move references outside. It is also applied to closures that don’t capture anything.

For ex.:

fn yo<T>(a: T ) where T: Fn() {

a();

}

accepts a closure that implements Fn trait.

Moving value out of closure means passing ownership of a capture reference to another method or simply passing ownership to some item outside the closure. For ex.:

let mut list=vec![String::from("2")];

let val=String::from("3");

list.push(val);

let p =val;

is an error as val doesn’t have ownership of its value anymore.

* + 1. Move closure: When a closure captures a variable, it copies it. Unless it has to modify it, then move is already applied. For ex.:

let mut a = Box::new(2);

let b = || {

\*a=3;

println!("{}", \*a);

};

println!("{}",\*a);

b();

is an error as println outside the closure can’t get a reference as a is moved into b.

We can explicitly specify move with

move|| {…}

move then moves the captured values into the scope.

* + 1. Return closures: Closures are represented by traits, meaning they are DSTs and hence can’t be returned normally by other functions. To solve this issue, we can use trait objects, i.e.,

fn yo() -> Box<dyn Fn(i32)->i32> {

Box::new(|x| x\*2);

}

works as the closure is now treated as a trait object and hence even though Rust has to figure out the size at runtime, it can return closures.

* 1. Iterators: Iterates over a sequence of values, iterators are lazy so they don’t iterate until the value is requested.

For ex.:

let list= vec![1,2,3];

let listIt = list.iter();

for item in listIt {

..//use item

}

* + 1. Iterator implementation: Internally iterators are defined as a trait which looks like this

pub trait Iterator {

type Item;

fn next(&mut self) -> Option<Self::Item>;

// methods with default implementations elided

}

The Item type is an associated type. It basically says that the type implementing this trait must define the type Item as well.

* + 1. next: An iterator has a next method which must be defined if we use the trait on our own type. The next method returns the current value and then changes the currently tracked item going to the next item in the sequence.

For ex.:

let list=…;

let mut listIt= list.iter();

println!(…, listIt.next());

println!(…, listIt.next());

println!(…, listIt.next());

…

Here we needed to make listIt variable mutable as it changes the internal state of the variable but we don’t need to do this when we use for loop as it takes ownership of the iterator and made it mutable automatically.

Many methods call next on iterators so this is why we need to manually define it if we implement its trait. Methods that call next() are called consuming adaptors, as calling next() consumes the iterator. For ex.:

let list=…;

let listIt=…;

let sumVal= listIt.sum(); //sum is a consuming adaptor that sums all the values in the sequence and returns value of type T where T is vec<T>.

* + 1. iter variants: iter() takes an immutable reference of all items. We can use into\_iter() to get owned values (i.e., the iterator takes ownership of the sequence and the values are owned by it) or iter\_mut() to get mutable reference of all items.
    2. Iterator adaptors: Methods defined on the Iterator trait that don’t consume the iterator. They allow us to change an iterator into different kinds of iterators. Because iterators are lazy, we need to call a consuming adaptor on iterator adaptors to actually run all the adaptors.

For ex.:

let list=…;

let listIt=…;

listIt.iter().map(|x| 2\*x);

is a warning as map is an iterator adaptor and hence won’t do anything unless any consumer pushes it.

let list2: Vec<\_> = listIt.iter().map(|x| 2\*x);

works, as it will be ran whenever list2 is accessed.

let list2: Vec<\_> = listIt.iter().map(|x| 2\*x).collect();

would drain the map right away as collect() is a consumer adaptor.

* + 1. Closures in iterator adaptors: Since closure capture values. Iterator adaptors that accept them become even more useful.

For ex.:

…

let y=2;

let list2= list.into\_iter().filter(|x| x==y).collect();

works. Here, using into\_iter() passes the ownership to the iterator and hence the filter method doesn’t need to create a new vector to return into list2.

* + 1. Iterators vs loops: Iterators follow the zero-cost abstraction principle in Rust. This basically means unlike loops, iterators will always be the most efficient ways of iterating over sequences. They will always be equal, if not faster, than raw loops. So it is recommended to use iterators wherever loops are concerned. This is because iterators are unrolled. Unrolling is an optimization that compilers perform by unwrapping fixed size loops into individual pieces of code, so if a loop runs 12 times then 12 similar definitions are created. This allows the code to be directly picked up by CPU registers and be much faster than waiting for a loop-controlling code.

“…zero-overhead principle: What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.”

* Bjarne Stroustrup

1. Reading env args: To do so,

use std::env;

…

let values: Vec<String>= env::args().collect();

//puts env args as String objects into values.

To pass env args,

cargo run arg1 arg2 …

Sometimes env args can fail due to Unicode issues. To solve that we can use args\_os() instead, this returns values of OsString type.

1. Reading env vars: To do so,

use std::env;

let val= env::var("MYENV\_VAR");

let isOk= val.is\_ok();

reads the given env var and returns Result in val. is\_ok() returns true if the Result is Ok().

1. Build Profiles: cargo build builds the dev build which is unoptimized but takes shorter to compile. cargo build –release builds in release mode which takes more compilation time but has more optimizations.
   1. To configure build profiles:

[profile.dev]

opt-level = 0

[profile.release]

opt-level = 3

in cargo.toml. The lines following [profile.\*] describe the build profile behavior. opt-level defines the optimization level which goes from 0 to 3 where 3 takes the longest to compile and gives the best performance. Default for dev is 0 and release is 3.

1. Publishing a crate: To publish a crate, first create an account on crates.io then get the API key and login with cargo login <key>

The package name must be unique.

We also need to define a license, we do both with

[package]

name = "guessing\_game"

version = "0.1.0"

edition = "2021"

description = "A fun game where you guess what number the computer has chosen."

license = "MIT OR Apache-2.0"

where license string can be gotten from <https://spdx.org/licenses/>.

We can use multiple with OR.

Then we publish with cargo publish.

Crates published to crates.io cannot be deleted, ever.

They can only receive later versions with cargo publish when our account is logged in and name has already been created by us where the version number is incremented based on rules from <https://semver.org/>. However, older versions will still persist.

* 1. yank: We can yank a version of a crate, disallowing new packages from depending on that version. Packages already using it aren’t affected.

To do so,

cargo yank --vers 1.0.1

and

cargo yank --vers 1.0.1 --undo

to unyank a version.

1. cargo install: Installs a binary to be used directly. By default cargo install installs a binary to $Home/.cargo/bin so we have to add that path to $PATH. Only packages on crates.io that are a binary can be installed like this.

For ex.:

cargo install ripgrep

installs rigrep, a rust package for searching files.

1. Pointers: Just like in C++, points to an address in memory. The most basic kind of pointer is a reference denoted with & . They don’t have any overhead and allow addresses to be passed around.
   1. Smart-Pointers: In Rust, smart pointers are special pointers that not only point to a piece of memory but also have some sort of metadata and additional functionality such as clearing memory when no one is accessing them. String and vector are smart pointers in Rust. Smart-Pointers implement the deref and drop trait at the very least.
   2. Reference types vs. normal types: Reference types (&i32) can’t be used with normal types (i32) because we need to first dereference their values to use them.
      1. Dereference operator: \*

For ex.:

let a=2;

let b = &a;

if(a==\*b)

{

…

}

works

\*var is just a shorthand form of

\*(var.deref())

The reason another \* exists is because it is a reference too. The deref method returns a reference and that avoids copy/move of the value needlessly. This isn’t an infinite reference loop, as internally the deref trait doesn’t use any more references.

* + 1. Custom type dereference: We can create our own custom types and get their values using deref op by implementing deref trait.

For ex.:

struct MyBox<T>(T);

impl<T> MyBox<T> {

fn new(x: T) -> MyBox<T> {

MyBox(x)

}

}

use std::ops::Deref;

impl<T> Deref for MyBox<T> {

type Target = T;

fn deref(&self) -> &Self::Target {

&self.0

}

}

type Target = T defines the associated type for the trait here.

When we call a value of type MyBox with \* then the fn deref is ran and the value in it is passed back, this is because \*var is actually a shorthand for \*(var.deref()). self.0 simply returns a reference to the first value of a tuple, which deref sees MyBox’s values as.

We can still dereference values with & however that has to be done manually and works only on references. When we implement the deref trait, the compiler can dereference normal values of the type as well and works on more than just references.

* + 1. Deref coercion: Rust can automatically convert a reference from type A to type B if type A implements the deref trait such that it returns type B. Like &String can be put into variables that need &str. Deref coercion is pretty powerful as Rust can even chain coercions. For ex.:

Using the MyBox type implemented above:

let myb= MyBox::new(String::from(“yee”));

let ya: &str= &myb

let omega: &str= &(\*m)[..]

works

MyBox gets converted to String which gets converted to &String which gets converted to &str. All because MyBox and String types implement deref trait and return values of type that the next function accepts. ‘omega’ shows how we would do it manually if Rust Coercion wasn’t a thing.

Deref coercion is solved at compile time, so there is no runtime cost.

* + 1. Mutable Deref: Just like Deref trait, we have use std::ops::DerefMut. This trait returns a mutable reference. Rust deref coercion expects the following:

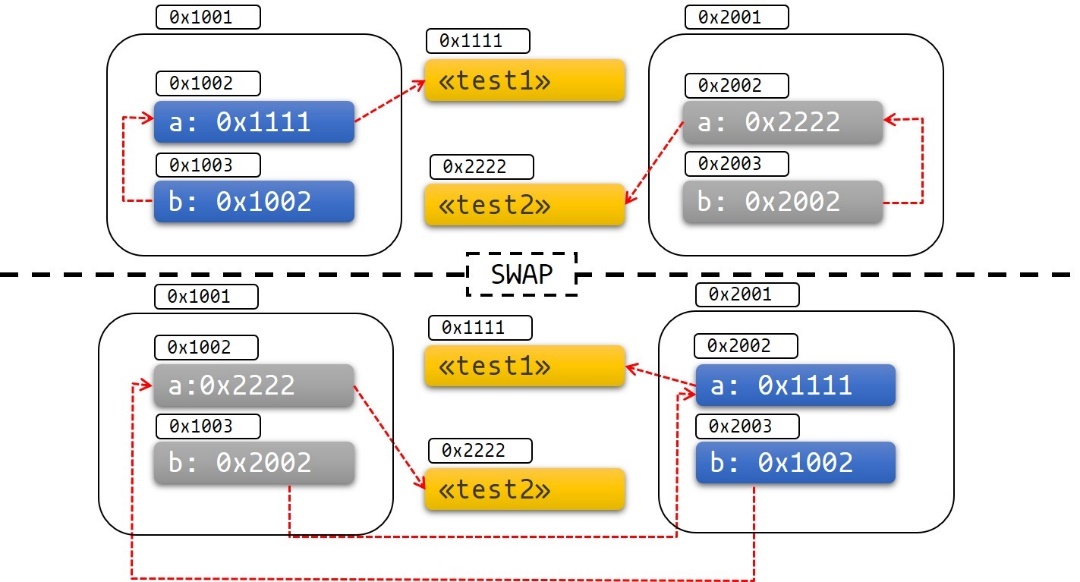
From &T to &U when T: Deref<Target=U>

From &mut T to &mut U when T: DerefMut<Target=U>

From &mut T to &U when T: Deref<Target=U>

All simply say immutable ref -> immutable ref, mutable ref-> mutable ref and mutable ref -> immutable ref. The last is true because a mutable reference takes ownership of the data which means it only has 1 reference to the data and hence it guarantees if the data changes no one is affected but the reverse is not true. Which means if a mutable ref wants to turn immutable then there’s no one to be affected by the change as immutability can be considered a subset of mutability.

* + 1. Pointers and the address problem:



When we store a pointer as a type and then do a mem swap with another of the same type then the address the pointer is pointing to is indeed moved. But, the values at the address theirselves are moved too, this means the pointer is pointing to a location in memory that’s now occupied by the swapped type.

For this reason extra care must be taken with pointers.

* 1. Drop Trait: Just like deref trait, drop trait is a trait that smart-pointers implement. This trait needs implementation for the drop method which takes a mutable copy of self and then runs some cleanup code. The drop method is called automatically by the Rust compiler when the scope of the type goes bust. Drop trait is included in the prelude. For ex.:

struct MyT{ yo:i32 }

impl Drop for MyT {

fn drop(&mut self) {

//do something with self.yo.

}

}

We don’t need to remove primitives by ourself. The drop method is aka destructor.

Variables in Rust are deallocated in the reverse order of their creation.

* + 1. Explicit drop: We can’t explicitly call the destructor for any type. This is to avoid the double free error which would happen as Rust automatically calls the destructor at the end of the scope. However, if we really need to explicity drop a value we use the std::mem::drop method. It is included in the prelude so,

let myVar=…;

drop(myVar);

works and drops the instance, if it implements the Drop trait then the traits drop() method is called as well.

Rust still ensures the validity of references and values at compile time using its ownership system so explicitly calling drop doesn’t cause any negative effects.

For primitives, drop does nothing.

* 1. Types of Smart-Pointers in Rust:
     1. Box<T>: Simple type with no overhead that stores values on the heap. For ex.:

let sBox= Box::new(5);

and access the value with sBox.

It automatically gets deallocated (along with the value) at the end of the scope.

* + - 1. Recursive Type: Boxes are useful in storing recursive type. This is a type whose value can have another value of same type as part of itself.

For ex.:

enum List {

Cons(i32, List),

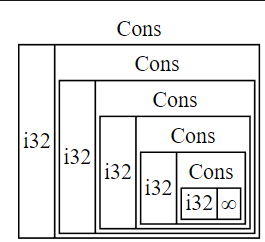
Nil,

}

Here Nil is just a value of type List.

let list= Cons(1,Cons(2,Cons(3,Nil)));

This is a data type in Lisp called Cons (constructor function). But it’d fail to compile in Rust as it can’t check it’s total size at compile time.

This is because when Rust tries to measure the size of an enum type, it looks at all its fields and sees which field takes the most size and that’s the size of the enum type. However, with a recursive enum, it’d need to check the same type infinitely.   


To fix this issue we can use Box<T> as the size of a pointer is known,

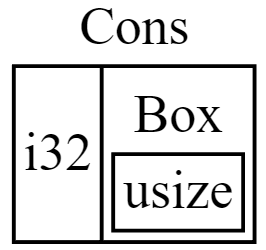
enum List{

Cons(i32, Box<List>),

Nil,

}

let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));



Box implements deref and drop trait.

* + - 1. We can dereference values with the deref op.
    1. Rc<T>: It is known as Reference Counter. It’s just like SharedPtr in C++ and keeps count of all its references. This allows multiple references to the same value on the heap.

For ex.:

use std::rc::Rc;

let a= Rc::new(10);

let b= Rc::clone(&a);

…

We use Rc::clone() instead of a.clone() as the former increments the reference counter by 1 (so it’d be 2 in this case as new start it at 1) and returns a reference but the latter deep-copies the value and doesn’t increase the counter.

b might go out of scope, in which case the count drops by 1.

Using Rc<T> allows a single value on heap to have multiple owners and Rc itself only goes out of scope when all the owners die. Rc<T> is only usable with immutable references.

* + - 1. strong\_count: Rc::strong\_count(<var>) where var could be like &a returns an i32 representing the count of references.

Only when the strong count hits 0 is the Rc dropped.

* + - 1. Weak<T>: Rc and RefCell together can lead to a recursive reference loop which means the strong count would never hit 0 and hence Rc will never be cleaned up.

However, Weak<T> doesn’t increase the strong count and hence the Rc can be dropped even if it has relying references. To do so,

let a= Rc::new(2);

let b= Rc::downgrade(&a);

let c= b.upgrade();

Here b gets a value of type Weak<i32>.

This means ‘a’ can get dropped even while b has a reference to it. upgrade upgrades the Weak to a Rc<T> reference again, however since the value may have been dropped it returns an Option<Rc<i32>>.

* + - 1. weak\_count: Just like strong\_count returns the strong counter, weak\_count returns the weak counter of an Rc.
      2. Almost all Rust types implement the Send trait, but Rc<T> doesn’t. This is because if a Rc<T> is cloned and then sent to a different thread, then both might try to update the same count for a new reference which poses concurrency problems.
    1. RefCell<T>: It uses the interior mutability design pattern of Rust. This design pattern allows immutable references to be able to mutate data internally. As this violates the ownership rules, internally it is a part of unsafe Rust. While unsafe code can be used normally, it requires the developer to ensure bugs don’t occur at runtime such as data races.   
       Rust is inherently conservative with its borrow rules. Sometimes even if a correct program is ran, it may not follow its rules and hence prevent Rust from compiling it. This is why unsafe code is allowed, as the checks are then passed over to runtime where if the code causes an issue, it panics and crashes the program.
       1. Using RefCell: We use it just like Rc<T>. RefCell::new(<value>) and then borrow mutable copies with .borrow\_mut() which returns a RefMut<T> object and can be used to modify the internal value.

The borrow checking rules are implemented at runtime for RefCell.

For example:

let a= RefCell::new(2);

let b= &a;

let mut c= b.borrow\_mut();

\*c=3;

works.

* + 1. Cell<T>: Just like RefCell but instead of references it internally creates new copies.
    2. Arc<T>: Just like Rc but uses Atomics.

1. Fearless Concurrency: Rust’s type system along with its ownership system is also effective against concurrency related issues. As such, Rust compiler can tackle most issues we may face with concurrency early on in compilation itself. This is the reason why Rust’s concurrency can be called fearless.
   1. Threads and Processes: A process is an individual program, whereas threads are part of the program that it has spun up on individual components. A single program is by default, single threaded.
      1. Spawn and sleep: To spawn a thread in Rust,

use std::{thread, time::Duration};

fn main(){

let aHandle= thread::spawn(|| {

println!("yo");

for i in 1..20 {

println!("{}",i);

}

});

println!("enddd");

thread::sleep(Duration::from\_millis(10));

}

‘aHandle’ will get a JoinHandle<()> type here, i.e. it owns the value. If we return a value from the closure, it will pick that type instead.

This prints   
enddd

yo  
1  
…

Threads are started by the OS scheduler so it is not certain when thread in b is spun up, however main thread doesn’t wait for these threads to finish up meaning the thread may still have work left if ran like this.

* + 1. Join: Calling join on a JoinHandle will start its thread and pause the thread calling join until the thread is finished doing its work. For ex.:

aHandle= …

aHandle.join().unwrap();

starts the thread in aHandle and blocks the main thread until aHandle thread finishes. Since join returns a Result<T,K>, we use the unwrap to get the value returned by the closure. If the thread failed for any reason, it returns an Err instead of an Ok.

* + 1. Move closure: If we use capture values in the closure, then it is a lifetime issue as the owner might die before the thread is even ran. To counter this, we can explicitly pass ownership to the closure and hence ensure the value’s lifetime. However, this means the value won’t be usable outside the closure. For ex.:

use std::thread;

fn main() {

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

let handle = thread::spawn(move || {

println!("Here's a vector: {:?}", v);

});

handle.join().unwrap();

}

works, but non-move closure wouldn’t work here.

* 1. Atomic types: Atomic types in std::sync::atomic are similar to their normal counterparts, like Rc<T> and Arc<T> but the difference is that atomic types use atomics, which ensure the data type can be passed between threads safely and internal state won’t be disturbed with multiple operations on it. Unlike their normal counterparts, atomics incur a performance penalty, which is why they are only used when concurrency is of paramount importance.
  2. Channels: Just like Go, Rust also believes in message passing to share memory between threads. A channel has an input end/transmitter end and a receiver end.

To create a channel, we use the std::sync crate.

For ex.:

std::sync::mpsc;

let (tx,rx)= mpsc.channe();

thread::spawn(move|| {

…

tx.send(22).unwrap();

});

let rcvd= rx.recv().unwrap();

println!(…, rcvd);

Here, (tx,rx) gets the transmitter and receiver types from mpsc.channel() where mpsc stands for multiple producer single consumer. tx sends a value in the channel and rx.recv() blocks the main thread until it it receives a value in its channel, i.e., in this case when this thread runs and sends the value in the channel. recv also returns a Result type. There’s also the non-blocking variant of recv, try\_recv() which doesn’t block and immediately returns Result.

Values given to send() have their ownership transferred to the channel. So if we pass a variable like send(value) then value’s ownership is passed to the channel (even without using references). send() returns a Resut, which tells if it was able to send the value. By using unwrap() we make sure it throws if it fails. We also, move the Send object as the thread needs ownership of the channel to work with it.

* + 1. Single channel Single receiver multiple receives:

use std::sync::mpsc;

use std::{thread, time::Duration};

fn main() {

let (a, b) = mpsc::channel();

thread::spawn(move || {

for i in 1..10 {

a.send(i).unwrap();

thread::sleep(Duration::from\_millis(10))

}

});

for i in b {

println!("{}", i);

}

}

Each iteration on receive gets an Ok(T) of the value in b and pauses the thread until send channel sends a value (it probably sends a value indicating the end as well) for all values, then it continues the main thread to continue executing.

* + 1. Single channel Single Reciever Multiple Recieves Multiple Sender:

mpsc allows us to clone() the sender. Meaning,

let a:Sender<i32>;

let b:Receiver<i32>;

(a,b)= mpsc::channel();

let c= a.clone();

is valid, and we can pass a and c to different threads whilst using the same receiver to receive the values sent in the channel.

* 1. Mutex: Stands for Mutual Exclusion. A Mutex is a type that allows only 1 entity to access data at any point of time by locking, other threads then ask the mutex for the lock and mutex puts them on halt until the thread inside has finished processing the data and unlocked the mutex. When using mutex every thread has to handle 2 things, acquiring mutex lock before working on the data and after working on the data unlocking the mutex.
     1. Mutex<T>: To create a mutex in Rust,

use std::sync::Mutex;

let a= Mutex::new(5);

{

let mut b= a.lock().unwrap();

\*b=3;

}

println!(“{}”,a.lock().unwrap());

Mutex::new(); creates a Mutex of type inferred from the value given to new and puts the value inside the Mutex.

lock() returns a Result where it gives Err if it was unable to acquire lock on the resource, then Ok() returns an object of type MutexGuard<T>. The MutexGuard is a smart pointer so it implements the drop and the deref trait, which allows us to modify the value inside. This is how we can get a mutable reference of an immutable value and hence modify it. At the end of the scope, when the smart pointer is dropped, the lock is released as well.

* + 1. Using Mutex across threads with Arc<T>: When we use threads, we have to move ownership of the captured types, this is why we need types like Arc to clone and pass the mutex around. For ex.:

let a= Arc::new(Mutex::new(3));

let b= Arc.clone(&a);

thread.spawn(move|| {

let mut c= b.lock().unwrap();

\*c= 5;

})

println!(“{}”, \*b.lock().unwrap());

works. But if had used Rc<T> then we would have a thread safety compile time error and if we didn’t use any then we wouldn’t be able to use the mutex again in the main thread.

* 1. Sync and Send trait: The send trait in the std::marker crate is a trait required for values of a type to be safely transferred between threads. Almost all Rust types implement the Send trait, but Rc<T> doesn’t. Similarly, the sync trait is a trait that defines if a type can be accessed from multiple threads. If any type T implements Send then &T also implements Send and if a type implements Send then it also implements Sync. Almost all primitive types are hence, Send and Sync trait implementations.
     1. These traits are known as Marker Traits, as if any type is made up of all the types that implement Send and Sync then it also implements Send and Sync.

This is why we don’t need to implement these traits generally, and if we do then that’s part of unsafe Rust.

* 1. Async: Rust supports the async/await model. The async model is 0 cost and even lazy, i.e., futures in rust are inert and only make progress when polled, otherwise they remain paused and when dropped stop immediately. Async code is statically dispatched. However, Rust does not include any runtime for async by default, this makes it necessary to use an external crate to be able to use async. This is a design choice as all async runtimes have their own pros and cons. This also means async binaries are bigger in size as they have to include their own runtimes along with async state machines. The alternative is using std::threads directly, which incurs a significant CPU/memory overhead if left unoptimized and is not ideal for general purpose concurrent programs. Async isn’t better than threads, it just has different uses.

Async syntax support and the Future trait are provided by the std itself. Traits don’t support async functions directly. Care must be taken with async code, as it works on runtime state machines which means some exceptions can silently pass compilation.

Async and sync Rust can work together but it generally turns the code more complex.

* + 1. Basic async crate:

futures = "0.3"

Basic async fn,

use futures::executor::block\_on;

async fn yo() {…}

and run it with

fn main() {

let yofn=yo(); //returns a type that implements Future trait

block\_on(yofn);

}

Every async program needs an executor to define execution policy for the async functions when run from a sync function. block\_on is like join on thread, it waits till the given future is ran to completion. Every async block implements the Future trait automatically.

* + 1. Async functions can call the await method on other async functions. myfn.await();. This will run the other fn concurrently.

They can also call the join macro,

futures::join!(fn1, fn2, fn3…);

This function runs the given async functions concurrently like await but it is used to call multiple of them at once. If an async method blocks then the thread is passed to the other method and so on until all are blocked then the executor gets to decide what to do with them.

* + 1. Future trait: The future trait’s working helps us understand async Rust better.

It basically looks like this,

trait SimpleFuture {

type Output;

fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;

}

enum Poll<T> {

Ready(T),

Pending,

}

For example:

pub struct SocketRead<'a> {

socket: &'a Socket,

}

impl SimpleFuture for SocketRead<'\_> {

type Output = Vec<u8>;

fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {

if self.socket.has\_data\_to\_read() {

// The socket has data -- read it into a buffer and return it.

Poll::Ready(self.socket.read\_buf())

} else {

// The socket does not yet have data.

//

// Arrange for `wake` to be called once data is available.

// When data becomes available, `wake` will be called, and the

// user of this `Future` will know to call `poll` again and

// receive data.

self.socket.set\_readable\_callback(wake);

Poll::Pending

}

}

}

Basically, Future returns a Poll enum from the poll function. The executor calls the poll function and the enum indicates if the future is complete or pending. The poll function gets a closure function ‘wake’ which tells the executor when it can wake and call the poll function again to forward the future towards completion. Using this functionality Future trait allows multiple Futures to be run in the same thread as when one is pending other is run and so on for all of them.

The actual future trait looks like so

trait Future {

type Output;

fn poll(

// Note the change from `&mut self` to `Pin<&mut Self>`:

self: Pin<&mut Self>,

// and the change from `wake: fn()` to `cx: &mut Context<'\_>`:

cx: &mut Context<'\_>,

) -> Poll<Self::Output>;

}

They used Pinned trait and the context closure here. The context closure allows identifying which function called the wake, otherwise it’s god’s plan as it is just a pointer.

A future is stored inside a Waker type, this type provides the wake function and future is also stored in this type. The executor receives this type and calls it a ‘Task’.

The wake function is essentially called through integration with an IO-aware system blocking primitive, such as epoll on Linux, kqueue on FreeBSD and Mac OS, IOCP on Windows, and ports on Fuchsia (all of which are exposed through the cross-platform Rust crate mio). These primitives all allow a thread to block on multiple asynchronous IO events, returning once one of the events completes.

* + 1. Manual async function: Can be defined like so:

async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {

// This `async` block results in a type that implements

// `Future<Output = u8>`.

async {

let x: u8 = foo().await;

x + 5

}

}

That is, we can define async blocks (these blocks implement the Future trait automatically as well)

* + 1. Lifetimes: Async functions need lifetimes just like normal ones.

// This function:

async fn foo(x: &u8) -> u8 { \*x }

// Is equivalent to this function:

fn foo\_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {

async move { \*x }

}

Which means async functions needing references as args depend on their lifetimes for their output’s lifetime. This also means they must be awaited while the lifetimes of their refs are still valid.

If we create refs inside a method, then returning async method means the ref will be invalid by the time it is used.

For ex.:

fn bad() -> impl Future<Output = u8> {

let x = 5;

borrow\_x(&x) // ERROR: `x` does not live long enough

}

fn good() -> impl Future<Output = u8> {

async {

let x = 5;

borrow\_x(&x).await

}

}

To circumvent this, we simply return the async method’s awaited value, as by this time it has captured the value and will not raise a lifetime error.

* + 1. async move: Just like move blocks but for async blocks.
    2. Sync types: As Futures shift between threads it is unadvised to use references of types such as Rc or RefCell or even the mutex from std::sync. The types must be future-aware for the optimal behavior, a lot of these types are introduced in the futures crate, such as futures::mutex.
    3. Pointer Address Problem: Since async Rust relies on shifting references around threads, it can suffer from this issue as well. This is the reason why it uses the Pinned Trait Markers.
    4. Pin Type: The Pin type wraps pointer types, guaranteeing that when the type T doesn’t implement the Unpin trait, the value pointed to by the pointer won’t be moved (T: !Unpin, here ! says Unpin is a marker trait). Futures created by async are marked with the Unpin trait.

Primitives can be moved in and out of Pin, this is because they have no problem being moved around, so Pin<&mut u8> behaves the same as &mut u8. Pin types can be defined simply with Pin<&T>.

Like Pin<Box<T>>, Pin<&mut T> etc.

Pin is defined in std::pin:Pin;

We can pin a value to stack with

let mut test1 = unsafe { Pin::new\_unchecked(&mut value) };

or with pin\_mut!(values…);

Pinning is an unsafe task.

This means when we copy/move test1 into another variable, it’s type will be Pin<&T> instead of &T.

For ex.:

let x= "yo";

pin\_mut!(x);

let y= x;

Here y’s type will be Pin<&mut &str>. It moves x into y and y is now wrapped in Pin’s safety.

To make our own types implement the Unpin trait,

use std::marker::PhantomPinned;

#[derive(Debug)]

struct Test {

a: String,

b: \*const String,

\_marker: PhantomPinned,

}

Pinning a value on the heap,

let mut x= Box::pin(value); //returns a Pin<Box<T>>

To create a pin on the heap using just &T,

we use the pin\_utils::pin\_mut! macro (to create a Pin<&mut T>).

defined in a crate call pin\_utils.

* + 1. Stream trait: Just like iterators, but async.

For ex.:

async fn send\_recv() {

const BUFFER\_SIZE: usize = 10;

let (mut tx, mut rx) = mpsc::channel::<i32>(BUFFER\_SIZE);

tx.send(1).await.unwrap();

tx.send(2).await.unwrap();

drop(tx);

// `StreamExt::next` is similar to `Iterator::next`, but returns a

// type that implements `Future<Output = Option<T>>`.

assert\_eq!(Some(1), rx.next().await);

assert\_eq!(Some(2), rx.next().await);

assert\_eq!(None, rx.next().await);

}

or

when used in a loop,

async fn sum\_with\_next(mut stream: Pin<&mut dyn Stream<Item = i32>>) -> i32 {

use futures::stream::StreamExt; // for `next`

let mut sum = 0;

while let Some(item) = stream.next().await {

sum += item;

}

sum

}

for loop doesn’t work with stream.

* + 1. Multiple future execution: We can execute multiple futures with join!(…) or try\_join!(…). The try variant just drops all futures if a future returns an Err.

Similarly, we have select!(…).

This one runs the futures and finishes when any one of the future completes.

For ex.:

use futures::{

future::FutureExt, // for `.fuse()`

pin\_mut,

select,

};

async fn task\_one() { /\* ... \*/ }

async fn task\_two() { /\* ... \*/ }

async fn race\_tasks() {

let t1 = task\_one().fuse();

let t2 = task\_two().fuse();

pin\_mut!(t1, t2);

select! {

() = t1 => println!("task one completed first"),

() = t2 => println!("task two completed first"),

}

}

Here, .fuse() fuses the future, meaning future won’t be called if it has finished. This is useful when the future may be polled too often, as when Poll::Ready is returned the state of the future may not be optimal to retry execution. Fused ensures just that. So if we run select many times, fused prevents them from restarting.

The basic syntax for select is <pattern> = <expression> => <code>

select! also accepts a default => and a complete => expr. default is run when no futures are complete and complete is run if all of them are. This means default returns the select immediately.

Pinning the future is important as the futures need to implement the Unpin trait, which is ensured by pin\_mut!. Select takes a mutable reference instead of moving the futures into it, this ensures it can be called again and again.

* + 1. ? and turbofish operator (::<>) : Await can return an Err if the future returns a Result, we use ? to return the Err just like in normal functions. However, async blocks can’t return the type and hence the type inference fails. This is a known issue, which is why we must explicitly specify the return type to at-least the Ok().

let fut = async {

foo().await?;

bar().await?;

Ok::<(), MyError>(()) // <- note the explicit type annotation here

};

1. OOP: OOP was coined by Alan Kay in 1967 for his object passing message to other object architecture in Simula. Rust isn’t strictly Object Oriented as it has its own quirks. However, we can still use OOP design pattern in Rust.
   1. Acc. to the Gang of Four book, OOP is:

“Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.”

By this defn Rust is indeed OOP as structs and enums which can contain data and impl blocks which can contain methods are able to fit in the defn.

Similarly, if encapsulation is the topic then Rust’s pub keyword to make items public (private is default) handles that aspect.

Lastly inheritance, is something Rust does not offer. We cannot define a child for a type. However, we can use traits and generics just like interfaces and generics in other languages and that fills that gap. Inheritance imposes its own bounds which can become problematic, and the most important aspect of it, i.e. Polymorphism, which is basically that an object of a type can be replaced with another of another typee if it shares the same behavior, is present in Rust with its trait system. This is why Rust has what would be known as bounded parametric polymorphism.

* 1. Trait Objects: Rust doesn’t have the concept of ‘Object’ as we don’t have a single Object that contains both fields and methods. However, trait Objects are a feature that can be called as Objects. They are just like generics but instead of holding only a single type, they can hold multiple types which they can check at runtime.

For ex.:

fn main() {

let bag = Bag {

stuff: vec![Box::new(1i32), Box::new(2u32)],

};

let bigger\_bag = BigBag {

stuff: vec![Box::new(1i32), Box::new(2u32)],

};

}

struct Bag<T: Tiffin> {

stuff: Vec<Box<T>>,

}

struct BigBag {

stuff: Vec<Box<dyn Tiffin>>,

}

trait Tiffin {}

impl Tiffin for i32 {}

impl Tiffin for u32 {}

is a compiler error. As bag’s type is resolved to be either i32 or u32 but not both.

However, the Trait object syntax ‘dyn <trait name>’ allows duck typing which is what dynamically typed languages like Python do, if it walks like a duck and quacks like a duck then it’s a fucking duck.

This enables dynamic dispatch, which is the opposite of static dispatch, which is what generics do in Rust. In dynamic dispatch, the compiler does not know which method to call at compile time, it only knows that the method will be resolved at runtime. This is where Rust uses pointers automatically and at runtime the pointers point to the right method. As all of this requires precious cpu time, it has a runtime cost. So trait objects should be used only when necessary.

1. Unsafe Rust: It’s a part of Rust that allows us more freedom with our code. Rust’s static analysis is conservative, i.e., it’d rather disallow some valid programs from running rather than allowing some invalid programs to run. By using unsafe Rust we can tell Rust explicitly that our program is valid and that we do it all at our risk. Unsafe Rust also exists because Rust targets low-level system programming and the hardware is inherently unsafe so certain tasks can only be done in an unsafe way.

We use unsafe Rust by declaring a block with unsafe keyword.

For ex.:

unsafe fn foo() {}

fn main() {

unsafe {

foo();

}

}

* 1. Unsafe Superpowers: These 5 things are what we can do in unsafe blocks, aka unsafe superpowers. Unsafe Rust doesn’t completely disable Rust’s static analysis, it just allows these superpowers to be used and Rust doesn’t check these as strictly.

They are:

* + 1. Dereference a raw pointer:
       1. Raw Pointer: Rust also has the concept of raw pointers just like C++, however they are a part of unsafe rust. They allow multiple mutable/immutable references to the same value, aren’t guaranteed to point to a valid memory, can be null and don’t implement automatc cleanup.

They are \*const T (for immutable pointer) and \*mut T (for mutable pointer) and the \* here is part of the type not the deref op.

For ex.:

let num=5;

let ptr= &num as \*const i32;

let ptr2= &num as \*mut i32;

and \*ptr or \*ptr2 to dereference the value. Raw pointers can be created in safe Rust. However, we have to use unsafe block to dereference the values inside.

* + 1. Call an unsafe function or methods
       1. We declare functions as unsafe with the unsafe keyword. Then the entire function body is unsafe. So to call the function, we have to call them from an unsafe block or function. For ex.:

unsafe fn yo() {}

unsafe {

yo()

}

It is advisable to not mark functions as unsafe, rather use unsafe blocks inside it wherever needed and then put manual checks in them for values. This way we create a safe abstraction over an unsafe code.

* + - 1. Extern: Rust also supports FFI (foreign function interface) where Rust code can call a function written in another language. We declare these external functions inside Rust within extern blocks and then call the functions using unsafe blocks. This is because other languages don’t employ Rust’s Safety system and hence they are always unsafe in Rust’s eyes.

For ex.:

extern "C" {

fn abs(input: i32) -> i32;

}

fn main() {

unsafe {

println!("Absolute value of -3 according to C: {}", abs(-3));

}

}

Here, “C” is the name of the ABI (Application Binary Interface) Rust will look in for the function. There are some ABIs that Rust guarantees will always be available for Rust compiler to use and C is one of them, so no extra code is needed for this example to be run.

Similarly, Rust code can be called from other languages too. In that case we specify the target ABI to the extern keyword and then define our method.

For ex.:

#[no\_mangle]

pub extern "C" fn call\_from\_c() {

println!("Just called a Rust function from C!");

}

Then we can take the pdb file in the target directory and link it to a C program and use it there. no\_mangle attribute disables name mangling for the given function. Name mangling is a feature that all languages do to shorten item names for smaller binary sizes.

* + 1. Access or modify a mutable static variable
       1. Global/static mutable variables can be modified in unsafe blocks.

For ex.:

static mut x : i32 =2;

…

unsafe {

x=3;

}

* + 1. Implement an unsafe trait:
       1. Unsafe traits are those traits that have some methods whose invariants the compiler cannot check. For them we need to use unsafe implementation blocks. Like if we were to implement the Send and Sync trait we would need to mark the implementations unsafe. For ex.:

unsafe trait X{…}

unsafe impl X for A {…}

* + 1. Access fields of unions:
       1. Unions are the unions from C. They are basically like Rust enums where only 1 single value can be active at a time but are written like structs. They are primarily used to FFI with unions in C and are a part of unsafe Rust as Rust can’t guarantee the active type by itself.

1. Macros: Macros are like functions except they aren’t. Macros are a way of writing code that writes other code, this is known as metaprogramming. Macros expand to produce more code. So when we use println!(…) we are calling a macro and even before the compiler interprets the code the macros are run and hence the println! gets replaced with relevant code. Macros have a bit more power than functions, they can take variable number of parameters with any type and are run at compile time but before the code is interpreted so they can be used in situtations like implementing traits on other types. Macros must be brought into scope before they can be called, unlike functions which can be defined anywhere and run anywhere. Macros are also needed as Rust doesn’t support Reflection.

Rust has 4 types of macros:

Declarative macros with macro\_rules!

And Procedural macros, which include #[derive], attribute like macros which define attributes on items and lastly function like macros which operate on specific arguments.

* 1. For ex.:

#[macro\_export]

macro\_rules! vec {

( $( $x:expr ),\* ) => {

{

let mut temp\_vec = Vec::new();

$(

temp\_vec.push($x);

)\*

temp\_vec

}

};

}

is a simplified version of vec! macro. Here, macro\_rules! is a type of macro that’s widely used in Rust aka as a Declarative Macro. Their syntax is similar to a match expression. The macro export attribute here tells that the macro should be brought into scope whenever its parent crate is brought into scope.

macro\_rules! has some strange edge cases which is why Rust’s second declarative macro which will come in future will deprecate it.

* 1. Procedural Macros: Unlike Declarative macros, these take some code as input and produce some code as output. There are 3 types of them, custom derive, attribute-like and function-like.

To define our own procedural macros we have to create a crate of a special type where the macro’s definition would reside. However we can create multiple procedural macros per crate.

General layout:

In src/lib.rs

use proc\_macro;

#[some\_attribute]

pub fn some\_name(input: TokenStream) -> TokenStream {

}

proc\_macro crate is included with Rust and defines TokenStream which is basically a piece of code. some\_attribute defines the type of procedural macro here.

* + 1. To create a proc macro,

First create a new crate, say hello\_macro,

crate new hello\_macro --lib

then do whatever with crate,

in our case we create

pub trait HelloMacro {

fn hello\_macro();

}

then to create the macro for it, cd into the hello\_macro, then

crate new hello\_macro\_derive --lib

this name syntax is important.

Then edit the cargo.toml of macro

[lib]

proc-macro = true

[dependencies]

syn = "1.0"

quote = "1.0"

Then in its lib.rs,

use proc\_macro::TokenStream;

use quote::quote;

use syn;

#[proc\_macro\_derive(HelloMacro)]

pub fn hello\_macro\_derive(input: TokenStream) -> TokenStream {

// Construct a representation of Rust code as a syntax tree

// that we can manipulate

let ast = syn::parse(input).unwrap();

// Build the trait implementation

impl\_hello\_macro(&ast)

}

We didn’t need to create 2 crates for just this code, we could have done it in a single macro crate only. syn creates a data structure for our TokenStream string.

Here impl… is the function working on syn to implement the trait,

fn impl\_hello\_macro(ast: &syn::DeriveInput) -> TokenStream {

let name = &ast.ident;

let gen = quote! {

impl HelloMacro for #name {

fn hello\_macro() {

println!("Hello, Macro! My name is {}!", stringify!(#name));

}

}

};

gen.into()

}

Then to use this macro and our custom crate into our crate,

hello\_macro = { path = "../hello\_macro" }

hello\_macro\_derive = { path = "../hello\_macro/hello\_macro\_derive" }

in the crate’s dependencies.

And to run it,

use hello\_macro::HelloMacro;

use hello\_macro\_derive::HelloMacro;

#[derive(HelloMacro)]

struct Pancakes;

fn main() {

Pancakes::hello\_macro();

}

* + 1. Attribute like macro: Defined similarly to custom derive like macros,

#[route(GET, "/")]

fn index() {

will be defined as

#[proc\_macro\_attribute]

pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {

* + 1. Function like macros:

Similarly,

#[proc\_macro]

…

for them and call with myMacro!(…);

* + 1. #[macro\_use] and #[macro\_export]: We can export macros in crates just like other items, this allows crate A that uses crate B to be able to use crate B’s macros. To do so, first a macro in crate B needs to have a macro with the attribute #[macro\_export] then crate A only needs to specify #[macro\_use] on module/crate declaration.

For ex.:

<https://doc.rust-lang.org/reference/macros-by-example.html>

1. Raw Identifiers: Allow us to use keywords for our items. For ex.

fn match(…){…}

invoking it is an error.

But with raw identifier syntax,

fn r#match(…){…}

will be called as r#match(…)

and works.

1. All operators and symbols: <https://doc.rust-lang.org/book/appendix-02-operators.html>
2. All derivable traits: PartialOrd, Ord, Hash, Default, Clone and Copy and Debug.