**Rust Quick Reference**

Table of Contents

[1. Overview 3](#__RefHeading___Toc5046_1917222798)

[1.1 Characteristics 3](#__RefHeading___Toc5048_1917222798)

[1.2 Developer tools in Rust 3](#__RefHeading___Toc5050_1917222798)

[1.3 Fields on Rust 4](#__RefHeading___Toc5052_1917222798)

[2. Rust Fundamentals 4](#__RefHeading___Toc5054_1917222798)

[2.0 Variables 4](#__RefHeading___Toc5056_1917222798)

[2.1 Data Types 4](#__RefHeading___Toc5058_1917222798)

[2.1.0 Boolean type 4](#__RefHeading___Toc5060_1917222798)

[2.1.1 Rust’s char type 4](#__RefHeading___Toc5062_1917222798)

[2.1.2 Integer type 5](#__RefHeading___Toc5064_1917222798)

[2.1.3 Arrays vs Vectors 6](#__RefHeading___Toc5066_1917222798)

[2.2 Traits 6](#__RefHeading___Toc5068_1917222798)

[2.3 Compiling 6](#__RefHeading___Toc5070_1917222798)

[2.4 Strings 7](#__RefHeading___Toc5072_1917222798)

[Slice Type(&str) 8](#__RefHeading___Toc5074_1917222798)

[2.4.1 Two types of strings: String vs %str (and the implications of not having this on C language) 8](#__RefHeading___Toc5076_1917222798)

[2.5 Functions & Methods 9](#__RefHeading___Toc5078_1917222798)

[Functions 9](#__RefHeading___Toc5080_1917222798)

[Functions with returning values: 9](#__RefHeading___Toc5082_1917222798)

[Methods 9](#__RefHeading___Toc5084_1917222798)

[2.6 Failures 9](#__RefHeading___Toc5086_1917222798)

[2.7 Scope {} 9](#__RefHeading___Toc5088_1917222798)

[2.8 Loops 9](#__RefHeading___Toc5090_1917222798)

[3. Stack & Heap 10](#__RefHeading___Toc5092_1917222798)

[4. Ownership 10](#__RefHeading___Toc5094_1917222798)

[4.1 Ownership Rules 10](#__RefHeading___Toc5096_1917222798)

[4.2 Interacting with data: Move if Heap, Copy if Stack, 10](#__RefHeading___Toc5098_1917222798)

[5. Reference (&) and borrowing 11](#__RefHeading___Toc5100_1917222798)

[5.1 The Rules of References 11](#__RefHeading___Toc5102_1917222798)

[5.2 Borrowing 11](#__RefHeading___Toc5104_1917222798)

[5.3 Mutable references 12](#__RefHeading___Toc5106_1917222798)

[5.4 Dangling references 12](#__RefHeading___Toc5108_1917222798)

[6. Structures 12](#__RefHeading___Toc5110_1917222798)

[Instance 13](#__RefHeading___Toc5112_1917222798)

[Struct update syntax 13](#__RefHeading___Toc5114_1917222798)

[6.1 Tuple structure 13](#__RefHeading___Toc5116_1917222798)

[6.2 Methods 13](#__RefHeading___Toc5118_1917222798)

[6.3 Associated functions 13](#__RefHeading___Toc5120_1917222798)

[6.4 Derived traits 14](#__RefHeading___Toc5122_1917222798)

[6.5 Where’s the -> Operator? 14](#__RefHeading___Toc5124_1917222798)

[7. Enums and pattern matching 14](#__RefHeading___Toc5126_1917222798)

[7.1 Option enum vs Null 15](#__RefHeading___Toc5128_1917222798)

[7.2 match 16](#__RefHeading___Toc5130_1917222798)

[7.3 if-let, concise control flow 17](#__RefHeading___Toc5132_1917222798)

[8. Managing growing projects with packages, crates and modules 17](#__RefHeading___Toc5134_1917222798)

[8.1 Packages and crates 17](#__RefHeading___Toc5136_1917222798)

[8.2 Defining modules to control scope and privacy 18](#__RefHeading___Toc5138_1917222798)

[Privacy 18](#__RefHeading___Toc5140_1917222798)

[Paths 19](#__RefHeading___Toc5142_1917222798)

[Using multiple items 19](#__RefHeading___Toc5144_1917222798)

[Aliasing 19](#__RefHeading___Toc5146_1917222798)

[8.3 External packages 19](#__RefHeading___Toc5148_1917222798)

[8.4 Separating modules into different files. 20](#__RefHeading___Toc5150_1917222798)

[9. Common collections 20](#__RefHeading___Toc5152_1917222798)

[9.1 Vectors 20](#__RefHeading___Toc5154_1917222798)

[Reading elements of vector 20](#__RefHeading___Toc5156_1917222798)

[Iterating a vector 21](#__RefHeading___Toc5158_1917222798)

[Using Enum and vector to store multiple types 21](#__RefHeading___Toc5160_1917222798)

[9.2 Hash Maps 21](#__RefHeading___Toc5162_1917222798)

[Updating a Hash Map 21](#__RefHeading___Toc5164_1917222798)

[Hashing function 22](#__RefHeading___Toc5166_1917222798)

[10. Error Handling 22](#__RefHeading___Toc5168_1917222798)

[10.1 Unrecoverable errors with panic! 22](#__RefHeading___Toc5170_1917222798)

[10.2 Recoverable errors with Result 23](#__RefHeading___Toc5172_1917222798)

[Matching Different Errors 23](#__RefHeading___Toc5174_1917222798)

[Propagating errors 24](#__RefHeading___Toc5176_1917222798)

[10.3 to panic! or not to panic! 25](#__RefHeading___Toc5178_1917222798)

[11. Generic Types, Traits and Lifetimes 25](#__RefHeading___Toc5180_1917222798)

[11.1 Generic Data Types 25](#__RefHeading___Toc5182_1917222798)

[Performance of code using generics 25](#__RefHeading___Toc5184_1917222798)

[11.2 Traits: Defining shared behavior 25](#__RefHeading___Toc5186_1917222798)

[Creating a Trait 25](#__RefHeading___Toc5188_1917222798)

[Implementing a Trait on a type 26](#__RefHeading___Toc5190_1917222798)

[Default implementations and override 26](#__RefHeading___Toc5192_1917222798)

[Traits as parameters 26](#__RefHeading___Toc5194_1917222798)

[Fixing Generic “largest” function with Traits 27](#__RefHeading___Toc5196_1917222798)

[11.3 Validating references with Lifetimes 28](#__RefHeading___Toc5198_1917222798)

[Borrow checker 28](#__RefHeading___Toc5200_1917222798)

[Generic lifetimes annotation 28](#__RefHeading___Toc5202_1917222798)

[Predictable Lifetime by elision rules 28](#__RefHeading___Toc5204_1917222798)

[The static lifetime 29](#__RefHeading___Toc5206_1917222798)

[12 Writing automated test 29](#__RefHeading___Toc5208_1917222798)

[12.1 How to write tests 29](#__RefHeading___Toc5210_1917222798)

[Creating test functions 29](#__RefHeading___Toc5212_1917222798)

[12.2 Controlling how test are run 29](#__RefHeading___Toc5214_1917222798)

[12.3 Test organization 29](#__RefHeading___Toc5216_1917222798)

# 1. Overview

## 1.1 Characteristics

* Precompiled just like C.
* zero-cost abstractions: higher-level features that compile to lower-level code as fast as code written manually.
* Includes the official building system & packet manager **Cargo**, allowing the user to control and build dependencies.
* **Snake case** as the conventional style for function and variable names. In snake case, all letters are lowercase and underscores separate words. E.g. variable\_name;

## 1.2 Developer tools in Rust

* **Cargo**: the included dependency manager and build tool(similar to Makefile but native), makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem.

Cargo has a mechanism that ensures you can rebuild the same artifact every build (Cargo.TOML). Cargo will use only the versions of the dependencies you specified until you indicate otherwise. To ignore manual versions or to upgrade a package use cargo update (-update\_only\_optional\_Packet) which will update dependencies to latest versions.

* **Rustfmt**: ensures a consistent coding style across developers.
* The Rust **Language Server**: powers Integrated Development Environment (IDE) integration for code completion and inline error messages.
* **Crate**: collection of Rust source code files (Basically a library).
* The ***Prelude*** is the most basic library that possesses the minimum functionalities that Rust need to import into every program, it includes traits of fundamental types, destructors and overloading, heap allocation, ownership, clone, comparison traits, generic conversions, iterators, heap allocated strings and vectors.

## 1.3 Fields on Rust

Command line tools, web services, DevOps tooling, embedded devices, audio and video analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even major parts of the Firefox web browser.

## 1.4 Compiling

|  |  |
| --- | --- |
| Instruction | Description |
| Cargo new {name} | Generates the packet manager folder for Cargo to manage your rust project |
| Cargo build  --release  --target {TRIPLET} | Compiles Rust program and creates an executable file in target/debug folder  - Compiles with optimizations (superfast code however is slower compilation time)  - Compiles for a target denoted by the TRIPLET. E.g. thumbv7m-none-eabi |
| ./target/debug/{Cargo\_File} | Runs the program stored on the Default path of the project, once Cargo builds the project. |
| Cargo run | Builds and runs the program in one step. |
| Cargo check | Check correct compiling without producing an executable (speed-up the process) |
| Use | Import library |
| cargo readobj --bin {project} -- -file-headers | can print the ELF headers to confirm that this is an ARM binary |
| Cargo-size --bin {project} --release -- -A | can print the size of the linker sections of the binary. |
| cargo fmt | Reformats your code according community code style |

Figure 1.4.1 Compiling your project

# 2. Rust Fundamentals

## 2.0 Variables

let -> Creates a variable.

**{}**  -> Curly brackets are the “format specifiers” (called with % in C) of Rust. In Rust they are just a place holder.

E.g. Print to console

a) println! ("x = {} and y = {}", x, y); // Prints the output to the screen.

b)let s = format! ("x = {} and y = {}", x, y);// Returns a String with the contents (but doesn’t print anything)

*There exist a lot of format specifiers for rust, for example:*

* *{:02X} – for individual bytes (change 2->4 for hexadecimals)*

E.g

print!("{} (U+{:04X}) ", c, c as u32);

$ cargo run --quiet -- "aimée" // a (U+0061) i (U+0069) m (U+006D) é (U+00E9) e (U+0065)

## 2.1 Data Types

Rust is a statically typed language, which means that it must know the types of all variables at compile time. The compiler can usually infer what type we want to use based on the value and how we use it. In cases when many types are possible, such as when converting a String to a numeric type using parse, we must add a type annotation.

### 2.1.0 Boolean type

In Rust, Booleans are one byte in size. Boolean type is specified by bool keyword: let f: bool = false;

Rust will not automatically try to convert non-Boolean types to a Boolean, it must be explicit and provide a Boolean value for safety. Note that if does not require parenthesis.

E.g. let number = 3;

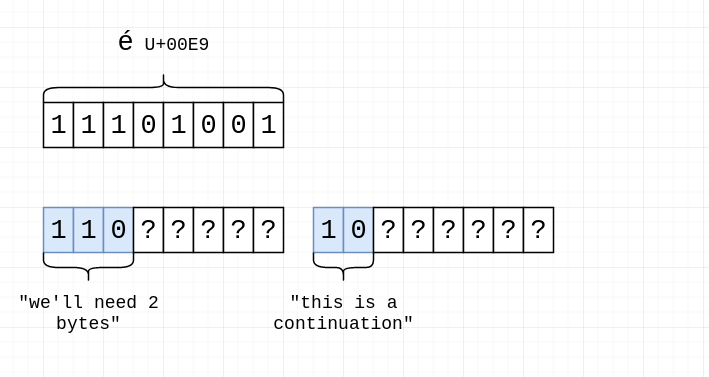
if number { println!("number was three"); } //Error: expected bool, found integer

### 2.1.1 Rust’s char type

Char is always 4 bytes in size as it represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII, like emojis. However not always the 4 bytes have information. In unicode encoding, the higher part indicates the #ofBytes needed to store the symbol ranging from 2 to 4 bytes:

* If a byte starts with 110 it means we’ll need two bytes
* If a byte starts with 1110 it means we'll need three bytes
* If a byte starts with 11110 it means we'll need four bytes
* If a byte starts with 10, it means it's a continuation of a multi-byte character sequence.

The following example encodes “é” by using 2 bytes(right align), leftover bits are filled with 0 (called padding).



F2.1.1.1 Encoding “é” in Unicode(? Represent a free slot)

### 2.1.2 Integer type

An integer is a number without a fractional component. It can be type annotated as follows:

let number = 13; //implicit data type  
let number: i32 = 13; //explicit data type

Signed numbers are stored using two’s complement representation.

|  |  |  |
| --- | --- | --- |
| **Integer Types** | | |
| **Length** | **Signed** | **Unsigned** |
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| arch | isize | Usize |
| Rust defaults integers to i32. | | |

Furthermore, you can use “**\_**” as a visual separator.

|  |  |
| --- | --- |
| **Number literals** | **Example** |
| Decimal | 98\_222 |
| Hex | 0xff |
| Octal | 0o77 |
| Binary | 0b1111\_0000 |
| Byte (u8 only) | b'A' |

Final note: **Integer overflow**

- In debug mode, Rust includes checks for integer overflow that cause your program to panic at runtime if an overflow occurs.

- In release mode, rust doesn’t include checks. So, if overflow occurs, Rust performs two’s complement wrapping.

### 2.1.3 Arrays vs Vectors

**Arrays** are useful when you want your data allocated on the stack rather than the heap or when you want to ensure you always have a fixed number of elements. E.g. let a: [i32; 5] = [1, 2, 3, 4, 5]; //array with type

An array isn’t as flexible as the vector type, though. A **vector** is a similar collection type provided by the standard library, it can grow or shrink in size(stored on heap).

#### Tuple

A tuple is a general way of grouping together several values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.

E.g. let tup: (i32, f64, u8) = (500, 6.4, 1);

Tuples can be destructured using patter matching and can be accessed using period “.”. E.g:

let five\_hundred = tup.0;

let six\_point\_four = tup.1;

let one = tup.2;

**Shadowing:** Shadowing lets us reuse variables with same name, rather than forcing us to create two unique variables. The second variable’s value is what appears when the variable is used. It’s also possible to change the type of the variable since we’re effectively creating a new variable. E.g. let x = 5; let x = x + 1; // x=6

## 2.2 Traits

--**mut**: Assigns mutable (modifiable, non-static content) attribute. In Rust variables are **immutable** by **default** to enforce safety and easy concurrency. E.g.: let mut guess = 5;

match number1.cmp(&number2) {

Ordering::Less => println!("Number 1 is smaller"),

Ordering::Greater => println!("Number 1 is larger!"),

\_ => (), //The “\_” pattern will match any value.

--**match:** allows us to compare a value against a series of patterns and then execute code based on which pattern matches. E.g. comparing returns an Ordering enum with 3 possible values: Less, Greater, Equal. Note that match is exhausting so all cases must be handled ( in case you want to handle remaining cases “\_” place holder is a useful to match any pattern)

--**const**: constants are ALWAYS immutable. Furthermore, constants require annotated type and can only be set to a constant expression computed in compile time. const MAX\_POINTS: u32 = 100\_000;

## 2.4 Strings

The **String** type is the most common type that has ownership over the contents of the string, it’s growable and UTF-8 encoded(Rust ensure this and panics if you try to put invalid UTF-8 symbols). Memory is requested from the memory allocator at runtime and returned(drop) via ownership system. To create a String(mutable) from a **string literal**(immutable, known and hardcoded into de executable at compile time (string literals are inside “” symbol)):

let mut s = String::from("hello"); //creates a String from a string literal “hello”  
s.push\_str(", world!"); // push\_str() appends a literal to a String  
println!("{}", s); // This will print `hello, world!`

Strings are hard because: propensity for exposing errors in compile time, complicated structure, and UTF-8.

Strings are implemented as a Collection of bytes, plus some methods that provide functionality to those bytes when interpreted as text. Rust has ONLY ONE string type in the core language which is STRING SLICE (str) that is usually seen in its borrowed form &str(reference to a UTF-8 data).

Indexing isn’t allowed on Strings because UTF-8 symbols may take more than 1 byte so, invalid character may show up if we tried to return only 1 byte as the following example:

let hello = "Здравствуйте";

let answer = &hello[0]; // Rust doesn’t compile indexing on Strings

**Iterating Over Strings**

You can access elements in a String in other ways, for example treating them as char or bytes.

|  |  |
| --- | --- |
| Interpreting Strings | |
| As chars | As bytes |
| for c in "न म स्ते".chars() {  print!("{}", c); // न म स स्ते  } | for b in "न म स्ते".bytes() {  println!("{},", b); // 224,164..135  } |

## 

### 2.4.1 Slice Type(&str)

(&str): it’s a slice pointing to an specific point of the binary; this is why string literals are immutable.

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. String slices allows indexing because (in contrast with Strings) they do not ensure UTF-8 symbols, instead they just reference bytes. This treatment is more general than String, so it allows Strings to be referenced as string slices(but losing UTF-8 symbol validity). Slice method from in String tracks a starting and an ending index. E.g.

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

let hello = &s[0..5];

let world = &s[6..11];

Slice Program**:** write a function that takes a string and returns the first word it finds in that string

fn first\_word(s: &String) -> &str {

let bytes = s.as\_bytes(); // Convert our String to an array of bytes

/\* iter is a method that returns each element in a collection and that enumerate wraps the result of iter and returns each element as part of a tuple instead \*/

for (i, &item) in bytes.iter().enumerate() {

if item == b' ' { //search for byte that represents the “space” (this uses byte literal).

return &s[0..i]; } }

&s[..] }

#### 2.4.2 Two types of strings: String vs %str (and the implications of not having this on C language)

https://fasterthanli.me/blog/2020/working-with-strings-in-rust/

Basically, this article says that in Rust String type values are always UTF-8 valid symbols because you get an error otherwise. It is heap-allocated.

&str refer to data from anywhere: heap, stack or even program’s data segment.

**The implications of not having String on C**

By comparison, C has no string type. It doesn't even have a real character type. C char is.. an ASCII character plus an additional bit - effectively, it's just a signed 8-bit integer: int8\_t.

* There is absolutely no guarantee that anything in a char\* is a valid UTF-8, or a valid something for that matter. There is no encoding associated to a char\*, which is just an address in memory. There is no length associated to it either, so computing its length involves finding the null terminator.
* Null-terminated strings are also a serious security concern. Not to mention that NULL is a valid Unicode character, so null-terminated strings cannot represent all valid UTF-8 strings.

## 2.5 Functions & Methods

#### Functions

fn example\_function(x: i32, y: i32) {

println!(“value of x is: {}”, x) …}

Rust doesn’t care where you define your functions, only that they’re defined somewhere (unlike C, where you must define a function before use it).

In function signatures, you must declare the type of each parameter, this means in the annotated form. E.g.

#### **Functions with returning values**:

Return type goes after an arrow (**->**). Functions can return early by using the return keyword and specifying a value, but most functions return the last expression implicitly, by not adding semicolon (this means that this is the return value).

fn plus\_one(x: i32) **->** i32 { x + 1 } //no semicolon, so it returns x + 1

plus\_one(5); //6

#### Methods

Methods are similar to functions, they can have parameters, return value

**object::method**() -> ´´method´´ is an associated function of an object type (static method). “::” is like namespace where 2 methods equally named are totally different if they namespace is different. E.g:

let guess = String::new(); //method that creates a new instance of a String

instance::method.submethod() -> Calls sub-method on method handle. E.g.: io::stdin().read\_line();

## 2.6 Failures

Result types are enumerations. For Result, the variants are Ok or Err. The Ok variant indicates the operation was successful, and inside Ok is the successfully generated value. The Err variant means the operation failed and Err contains information about how or why the operation failed. The purpose of these Result types is to encode error-handling information.

E.g.: .expect(“failed”) -> Expect Unwraps a result, yielding the content of an [Ok](https://doc.rust-lang.org/std/result/enum.Result.html#variant.Ok). Otherwise panics and includes message and content of Err.

## 2.7 Scope {}

A scope is the range within a program for which an item is valid.

## 2.8 Loops

|  |  |  |
| --- | --- | --- |
| **For** | **While** | **Loop** |
| Increased safety of the code and eliminated the chance of bugs that might result from going beyond the end of the array or not going far enough and missing some items. | Useful to evaluate a condition within a loop. However, is slower because the compiler adds runtime code to perform the conditional check on every element on every iteration through the loop. | Executes a block of code over and over again forever or until you explicitly tell it to stop. |

# 3. Stack & Heap

**Stack-allocated data has a known, fixed size, LIFO**. Data with an **unknown size at compile time or a size that might change must be stored on the heap instead.** The heap is less organized: when you put data on the heap, you request a certain amount of space. The operating system finds an empty spot in the heap that is big enough, marks it as being in use, and returns a pointer, which is the address of that location. This process is called allocating on the heap and is sometimes abbreviated as just allocating. Pushing values onto the stack is not considered allocating. Because the pointer is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you must follow the pointer.

# 4. Ownership

All programs must manage the way they use a computer’s memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it’s running.

## 4.1 [Ownership Rules](https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-rules)

* Each value in Rust has a **variable** that’s called its **owner**.
* There can only be **one** owner at a time.
* When the **owner** goes **out of scope**, the value will be dropped hence the memory is freed.

Note: Ownership rules applies to variables as well as functions.

## 4.2 Interacting with data: Move if Heap, Copy if Stack,

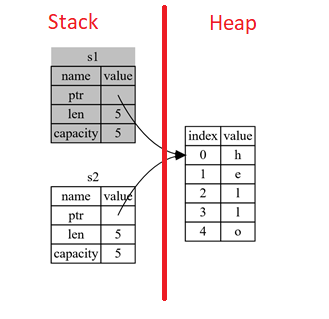
Rust will never automatically create “deep” copies of your heap allocated data(because could result in poor performance). Instead Rust performs a “move” operation, on which:

* Stack-allocated Data is Copied with an internal clone.
* Heap-allocated Data copies just the reference to the same location. It transfers ownership! So previous variable is **invalidated**(avoiding double free problem)**.**

This way, the previous owner will not try to drop memory when goes out of scope because it’s invalid; the new owner will now oversee the value dropping. In the next example, known size values like length and capacity are stored on the Stack so they are copied; dynamic values like String ptr data is allocated on heap so only the pointer is copied, however ownership is passed as shown on F4.1.1.

let s1 = String::from("hello"); //String stores some values on Heap and some on Stack

let s2 = s1; //s1 is now invalid for safety. Now S2 has ownership



F4.1.1 Representation in memory after s1 has been invalidated.

|  |  |
| --- | --- |
| Ownership in functions | |
| Pass **copy** | Pass **ownership** |
| // x is created on **stack**  let x = 5;  // x (i32) is Copied into function  makes\_copy(x);  //x is still valid here | // s is created on the **heap**  let s = String::from("hello");  //s pass ownership to the function...  take\_ownership(s);  //s is no longer valid here |

Note: To create a deep copy, including heap data, use clone method. let s2 = s1.clone();

# 5. Reference (&) and borrowing

**& ->** Get memory address: operator that gets the memory address (in hexadecimal) of a piece of data.

## 5.1 The Rules of References

* At any given time, you can have *either* one mutable reference *or* any number of immutable references.
* References must always be valid(lifetime).

## 5.2 Borrowing

It’s also possible to use a value without taking ownership, by using references (&), we call having references as function parameters **borrowing**. The scope in which the variable is valid isn’t affected by the borrowing variable/function, so we don’t have to drop what the reference points to when it goes out of scope because we **didn’t had ownership** in the first place.

{

let s1 = String::from("hello"); //s1 has ownership

let len = calculate\_length(&s1); //len borrows s1, and uses it.

// s1 keeps having ownership

} //s1 goes out of scope so is freed.

fn calculate\_length(s: &String) -> usize { s.len()}

Graphical user interface

Description automatically generated

5.1.1 Variable s borrowing variable s1. These & are references, and they allow you to refer to some value without taking ownership of it.

The opposite of reference is dereference, which is accomplish with dereference operator (\*) on chapter 15.

## 5.3 Mutable references

Note that **references are immutable by default**, to create a mutable reference, just add &mut s trait.

A big restriction on mutable references is that you can have **only ONE mutable reference** to a value in the **same scope,** also is not possible to have mutable and immutable references in the same scope. This restriction allows mutation in a very controlled fashion, avoiding race conditions, simultaneous access to the same piece of data, synchronization problems and sudden changes.

## 5.4 Dangling references

if you have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does, at compile time!.

# 6. Structures

Like tuples, structures can group many data types together. However, structures doesn´t rely on of data order declaration. Structs own its data so data referenced by a struct is valid as long s a struct is.

struct User {

username: String,

email: String,

sign\_in\_count: u64,

active: bool,

}

#### Instance

Instances are declared using key:value pairs, luckily there is a shorthand notation when variables and fields have the same Name (email: email -> email). Access to a specific field is granted through dot notation. Entire instance should be mutable, Rust does´t allow to mark only certain fields as mutable.

fn build\_user(email: String, username: String) -> User {

User {

email,

username,

active: true,

sign\_in\_count: 1,

}

}

#### Struct update syntax

It is possible to create a new instance that uses values of an existing one. And there is also a shorthand notation presented to set the remaining fields with another instance using “..” .

let user2 = User {

email: user1.email,

username: user1.username,

..user1 //remaining field on user2 are the same as user1.

};

Note: to update struct use dot notation. E.g: user1.email = String::from(“etapia@gmail.com”);

## 6.1 Tuple structure

Tuple structs have the added meaning the struct name provides, but don’t have names associated with their fields.

struct 3DPoint(i32, i32, i32);

let origin = 3DPoint(0, 0, 0);

## 6.2 Methods

Methods only differ from functions because they are defined within the context(impl) of the struct and their first parameter is always self, which represents the instance of the struct the method is being called on.

impl Rectangle {

fn area(&self) -> u32 { //add &mut self if you want to modify the struct

self.width \* self.height

}

}

Let rect1 = Rectangle { width: 30, height:50,}; //creating a Rectangle instance

Rect1.area(); //calling the area method

## 6.3 Associated functions

We’re allowed to define functions within impl blocks that don’t take self as a parameter, these are called **associated functions**(still functions, not methods)**.** Associated functions are often used for constructors that will return a new instance of the struct. To call this associated function, use the namespace“::” notation. E.g.

impl Rectangle {

fn square(size: u32) -> Rectangle { //no self

Rectangle { width: size, height: size }

}

}

let sq = Rectangle**::**square(3);

## 6.4 Derived traits

The println! macro can do many kinds of formatting, and by default, the curly brackets tell println! to use formatting known as Display (output intended for direct end user consumption). For primitive types we’ve seen so far there’s only one way you’d want to show them. But with structs, the way println! should format the output is less clear because there are more display possibilities: Do you want commas or not? Do you want to print the curly brackets? Should all the fields be shown? Due to this ambiguity, Rust doesn’t try to guess what we want, and structs don’t have a provided implementation of Display.

println!("rect1 is {**:?**}", rect1);. Putting the specifier **:?** inside the curly brackets tells println! we want to use an output format called Debug. The **Debug trait** enables us to print our struct in a way that is useful for developers so we can see its value while we’re debugging our code.

Rust does include functionality to print out debugging information, but we have to explicitly make that functionality available for our struct. To do that, we add the annotation #[derive(Debug)] just before the struct definition.

#[derive(Debug)]

struct Rectangle {

width: u32,

height: u32,

}

fn main() {

let rect1 = Rectangle { width: 30, height: 50 };

println!("rect1 is {:?}", rect1);

}

## 6.5 Where’s the -> Operator?

In C and C++, two different operators are used for calling methods: you use “.” if you’re calling a method on the object directly and -> if you’re calling the method on a pointer to the object and need to dereference the pointer first. In other words, if object is a pointer, object->something() is similar to (\*object).something().

Rust doesn’t have an equivalent to the -> operator; instead, Rust has a feature called **automatic referencing and dereferencing**. Here’s how it works: when you call a method with object.something(), Rust automatically adds in &, &mut, or \* so object matches the signature of the method. In other words, the following are the same: p1.distance(&p2); == (&p1).distance(&p2);

# 7. Enums and pattern matching

Enumerators allow us to enumerate all possibilities but only choosing one. It has some properties of structs as all variants should be treated as the same type.

|  |  |
| --- | --- |
| Simple Enum and usage | Data embedded in the Enum(using tuple) |
| enum IpAddrKind {  V4,  V6,  } | enum IpAddr {  V4(u8, u8, u8, u8),  V6(String),  } |
| let four = IpAddrKind::V4;  let six = IpAddrKind::V6;  \*Note that the variants of the enum are namespaced under its identifier, allowing either: V4 or V6.  fn route(ip: IpAddrKind) { } | let home = IpAddr::V4(192, 168, 50, 1);  let loopback = IpAddr::V6(String::from("::1"));  Note: you can put any kind of data inside an enum variable. |

enum Message {

Quit,

Move { x: i32, y: i32 },

Write(String),

ChangeColor(i32, i32, i32),

}

impl Message {

fn call(&self) {

// method body would be defined here

}

}

let m = Message::Write(String::from("hello")); //::Write is namespaced in Message enum

m.call();

**Using methods and associated functions on Enums**

Just as we’re able to define methods on structs using impl, we’re also able to define methods on enums. E.g.

## 7.1 Option enum vs Null

The concept that null is trying to express is a useful one: a null is a value that is currently invalid or absent for some reason. Despite being implemented poorly on C, where a null has led to innumerable errors, vulnerabilities, and system crashes. The problem with null values is that if you try to use a null value as a not-null value, you’ll get an error of some kind. This null property is pervasive because it’s extremely easy to make this kind of error.

The problem radicates on the implementation. Rust **does not have nulls**, but it does have an enum that can encode the concept of a value being present or absent. This enum is Option<T>, and it is defined by the standard library(prelude) as follows:

enum Option<T> {

Some(T),

None,

}

let some\_string = Some("a string"); //presence

let absent\_number: Option<i32> = None; //abscense

Option is better than Null because Option<T> and T (where T can be any type) are different types, the compiler won’t let us use an Option<T> value as if it were definitely a valid value. In order to use an Option<T> value, you want to have code that will handle each variant of Option Enum.

let y: Option<i8> = Some(5);

let sum = 8 + y; //Will not compile. Needs to handle Option<i8> Enum

\*Option is strong with match to handle cases. In other word, you have to convert Option<T> to a T before perform operations with T.

## 7.2 match

match is a control flow operator that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards and more.

enum Coin {

Penny,

Nickel,

Dime,

Quarter,

}

fn value\_in\_cents(coin: Coin) -> u8 {

match coin {

Coin::Penny => 1,

Coin::Nickel => 5,

Coin::Dime => 10,

Coin::Quarter => 25,

}

}

* =>, separates the pattern to match and the code to run, in this case the code is just the value.
* \_, wildcard that matches any value.

**Match with Option<T>:** function that takes an Option<i32> and, if there’s a value inside, adds 1 to that value. If there isn’t a value inside, the function should return the None value and not attempt to perform any operations.

fn plus\_one(x: Option<i32>) -> Option<i32> {

match x {

None => None,

Some(i) => Some(i + 1),

}

}

let five = Some(5);

let six = plus\_one(five);

let none = plus\_one(None);

Note: matches are exhaustive, all cases should be handled.

## 7.3 if-let, concise control flow

If let allows us to combine if and let into a less verbose way to handle values that match one pattern while ignoring the rest. Using if let means less typing, less indentation, and less boilerplate code. However, you lose the exhaustive checking that match enforces.

|  |  |
| --- | --- |
| Verbose matching pattern, only 1 case is useful | Shorthand if let, handles neatly 1 pattern |
| let some\_u8\_value = Some(0u8);  match some\_u8\_value {  Some(3) => println!("three"),  \_ => (),  } | if let Some(3) = some\_u8\_value {  println!("three");  } |

# 8. Managing growing projects with packages, crates and modules

As a project grows, you can organize code by splitting it into multiple modules and then multiple files. As a package grows, you can extract parts into separate crates that become external dependencies.

Once you’ve implemented an operation, other code can call that code via the code’s public interface without knowing how the implementation works.

Rust has features that allow you to manage your code’s organization, including which details are exposed(public), which details are private, and what names are in each scope in your programs. These features, sometimes collectively referred to as the module system, include:

* Packages: A Cargo feature that lets you build, test, and share crates. A package can contain multiple binary crates and at most one library crate.
* Crates: A tree of modules that produces a library or executable
* Modules and use: Let you control the organization, scope, and privacy of paths(private/public)
* Paths: A way of naming an item, such as a struct, function, or module

## 8.1 Packages and crates

When we create a new project via cargo new, Cargo creates the Cargo.toml file giving us a package. there’s no mention of src/main.rs because Cargo follows a convention that src/main.rs is the crate root of a binary crate with the same name as the package. Likewise, Cargo knows that if the package directory contains src/lib.rs, the package contains a library crate and src/lib.rs is its crate root. Cargo passes the crate root files to rustc to build the library or binary. Finally because crates are namespacedm which means that they have their own scope so no confusion of which library function is being called.

## 8.2 Defining modules to control scope and privacy

To structure our crate, rust allows organization via nested modules. E.g: The restaurant library. In the restaurant industry, some parts of a restaurant are referred to as front of house and others as back of house. Front of house is where customers are; this is where hosts seat customers, servers take orders and payment, and bartenders make drinks. Back of house is where the chefs and cooks work in the kitchen, dishwashers clean up, and managers do administrative work.

Create a new library named restaurant by running cargo new --lib restaurant;. Then define the module as:

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

fn seat\_at\_table() {}

}

mod serving {

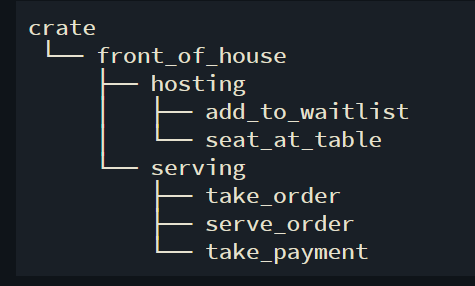
fn take\_order() {}

fn serve\_order() {}

fn take\_payment() {}

}

}



Note: Src/main.rs and src/lib.rs are called crate roots because the contents of these two form the root of the crate module structure/tree. Modules are like filesystem’s directory tree on a computer.

### Privacy

The way privacy works in Rust is that all items (functions, methods, structs, enums, modules, and constants) are private by default, which hides the inner implementation details and allow to control inner and outer code.

Pub keyword makes items public to the ancestor module.

Example:

mod back\_of\_house {

pub struct Breakfast {

pub toast: String,

seasonal\_fruit: String,

}

impl Breakfast {

pub fn summer(toast: &str) -> Breakfast {

Breakfast {

toast: String::from(toast),

seasonal\_fruit: String::from("peaches"),

}

}

}

}

pub fn eat\_at\_restaurant() {

// Order a breakfast in the summer with Rye toast

let mut meal = back\_of\_house::Breakfast::summer("Rye");

// Change our mind about what bread we'd like

meal.toast = String::from("Wheat");

println!("I'd like {} toast please", meal.toast);

// The next line won't compile if we uncomment it; we're not allowed

// to see or modify the seasonal fruit that comes with the meal

// meal.seasonal\_fruit = String::from("blueberries");

}

#### Paths

To find an item in a module tree, we use paths to navigate the filesystem. A path can take two forms:

* An absolute path starts from a crate root by using a crate name or a literal crate. E.g:
  + crate::front\_of\_house::hosting::add\_to\_waitlist();
* A relative path starts from the current module and uses self, super, or an identifier in the current module.

Use keyword brings a path into scope to use them as if they were local items. E.g:

use crate::front\_of\_house::hosting;

pub fn eat\_at\_restaurant(){ hosting::add\_to\_waitlist(); }

#### Using multiple items

Importing multiple items like:

use std::cmp::Ordering;

use std::io;

can be shorted as:

use std::{cmp::Ordering, io};

Operator \* specify all public items within a path.

#### Aliasing

As keyword allows aliasing paths to avoid conflict of 2 same named paths. E.g. use std::io::Result as IoResult;

## 8.3 External packages

Pulling external packages from crates.io involves:

1. Listing the packages in the Cargo.toml file
2. Bring the items into scope via use keyword

E.g:

|  |  |
| --- | --- |
| 1) On Cargo.toml | 2) On src/main.rs |
| [dependencies]  rand = "0.5.5" | use rand::Rng;  fn main() {let secret\_number = rand::thread\_rng().gen\_range(1, 101);} |

Note: std is an external package but its shipped with Cargo so no need of listing it on the Cargo.toml

## 8.4 Separating modules into different files.

When modules get large, you might want to separate the module with its own file. To do this:

1. Put a semicolon after mod front\_of\_house instead of a block, this tells Rust to load the contents of the module from another file with the same name as the module.
2. Create the src/front\_of\_house directory and file src/front\_of\_house/hosting.rs to contain definitions of hosting module.

|  |  |
| --- | --- |
| **On src/lib.rs** | **On** src/front\_of\_house.rs |
| mod front\_of\_house;  pub use crate::front\_of\_house::hosting;  pub fn eat\_at\_restaurant() {  hosting::add\_to\_waitlist();  **}** | pub mod hosting; //declaration of module |
| **On** src/front\_of\_house/hosting.rs |
| pub fn add\_to\_waitlist() {} |

# 9. Common collections

Standard library includes data structures called collections which may contain multiple data types. Unlike built-in array and tuples, these collections are stored on the heap!:

* Vector: allows you to store a more than one value of the same type in contiguous memory.
* String: is a collection of characters. We’ve mentioned the String type previously, but in this chapter we’ll talk about it in depth.
* Hash map: allows you to associate a value with a particular key. It’s a particular implementation of the more general data structure called a map.

## 9.1 Vectors

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

let third: &i32 = &v[2];

println!("The third element is {}", third);

match v.get(2) {

Some(third) => println!("The third element is {}", third),

None => println!("There is no third element."),

}

let v = vec![100, 32, 57];

for i in &v {

println!("{}", i);

### 9.1.1 Reading elements of vector

* Vectors are indexed by number, starting at zero.
* Two ways of accessing an element:
  + Using & and [], which give us a reference.
  + Using get method with the index passed as an argument, which give us an Option<&T>

let mut v:Vec<i32> = Vec::new(); //Vec<T>, generic so it can hold any type

v.push(5); // updating a vector  
//or

let v = vec![1, 2, 3]; // macro to create vector with initial values

### 9.1.2 Iterating a vector

We can also iterate over mutable references in order to change the elements.

let mut v = vec![100, 32, 57];

for i in &mut v {

\*i += 50; } // dereference operator (\*) to get to the value in i

#### Using Enum and vector to store multiple types

This makes use of the property that the variants of an enum are defined under the SAME ENUM TYPE :OO. However the disadvantage is that types have to be known at compile time(even thought is stored on heap) or use a trait object discussed on chapter 17.

enum SpreadsheetCell {

Int(i32),

Float(f64),

Text(String),

}

let row = vec![

SpreadsheetCell::Int(3),

SpreadsheetCell::Text(String::from("blue")),

SpreadsheetCell::Float(10.12),

## 9.2 Hash Maps

Hash maps are useful to look up data without using idex, but keys. Iterators may also be used to generate keys easily.

use std::collections::HashMap;

let mut scores = HashMap::new();

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

scores.insert(String::from("Yellow"), 50);

### 9.2.1 Updating a Hash Map

Although the number of key-values is growable, each key can ONLY have ONE VALUE associated at any time. You can handle collisions in three ways:

|  |  |  |
| --- | --- | --- |
| Collision Handling in hash maps | | |
| (default) Overwriting value. Using insert(). | Only updating value if key has no value. Using or\_insert(). | Updating a value based on Old value. Take the previous value and run and run an expression using it. |

#### Hashing function

By default, HashMap uses a “cryptographically strong” hashing function called Blake(see https://www.131002.net/siphash/siphash.pdf). Provides resistance to DoS attacks, but its tradeoff for security drops a little speed. However, it can be changed for a different hasher.

# 10. Error Handling

Rust divides errors(Rust doesn’t have exceptions) into 2 categories:

* Recoverable errors: its reasonable to report the problem to the user and retry the operation. it uses Result<T, E> enum
* Unrecoverable errors: symptoms of bugs, like trying to access a location beyond the end of an array. It uses panic! Macro to print a failure message, unwind and clean up the stack, finally exiting the program.

Errors can be expanded using --explain parameter or using --verbose.

## 10.1 Unrecoverable errors with panic!

A full panic! Unwinds and cleans the stack. Alternatively, Rust can just use abort profile(panic = ‘abort’) which ends the program without cleaning the stack to speed up the process and reduce binary size.

**Backtracing a panic!, and its C counterpart**

Lets use an accessing out-of-bound element.

In C, attempting to read beyond the end of data structure results in undefined behavior. You might get whatever is at the location in memory(buffer overread which leads to security vulnerabilities),

In Rust, the execution panics. The error points at the slice(libcore/slice/mod.rs) implementation of Rust source code and backtraces from there(shows all the function calls up to that point).

## 10.2 Recoverable errors with Result

use std::fs::File;

fn main() {

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

let f = match f {

Ok(file) => file, //when result is Ok, return the inner file value of the Ok

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

};

}

Most errors aren’t serious enough to require executing exit. For example, trying to open a file is better to recover from it by using Result<T, E> enum:

#### Matching Different Errors

The previous code panic in any case. However, If we want to instead open a file if the failure reason was due to inexistence of the file and panic in any other case, then we would simply divide the match expression.

use std::fs::File;

use std::io::ErrorKind;

fn main() {

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

let f = match f {

Ok(file) => file, //when result is Ok, return the inner file value of the Ok

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

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

Ok(fc) => fc, // returns the recently created file

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

},

other\_error => {

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

}

},

};

}

This way we can resolve for any specific or set of errors. Note that an error handler for unable to create new file had to be added as it might also fail. That’s a lot of match!, match is very useful but also very primitive, see closures on chapter 14. Closures would create something like this:

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);

}

});

}

‘unwrap’ and ‘expect’: shortcuts for Panic on Error

Unwrap is a shortcut method for match-Result, returns value inside Ok or panic! in case Err.

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

expect is similar to unwrap but let us choose the panic! error message.

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

#### Propagating errors

use std::fs::File;

use std::io;

use std::io::Read;

// function that returns a String inside Ok or err of type ioError

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

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

let mut f = match f {

Ok(file) => file,

Err(e) => return Err(e),}; //error prone call

let mut s = String::new();

match f.read\_to\_string(&mut s) {

Ok(\_) => Ok(s),

Err(e) => Err(e), //error prone call

}

}

When calls inside a function might fail, you can instead propagate the error to have more control over calling function. This is, return the error to the function whose contents has error prone calls.

There is also a shortcut for propagating errors: The **?** operator.

? operator can only be used on functions that return Result enum.

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

let mut s = String::new();

f.read\_to\_string(&mut s)?;

## 10.3 to panic! or not to panic!

Unwrap and expect are handy when PROTOTYPING, they act as placeholder for when you actually decide how to handle the errors in a more robust way. But they are not recommended for releases because they just exit the program.

When you know result will have an Ok value its fine to use unwrap. For example, hardcoded values.

Its advisable to panic! when:

* You could end up in a bad state(not expected).
* Input values are not valid. However, you can ensure data validity by creating Custom Types : pub struct Guess { value: i32, }.

# 11. Generic Types, Traits and Lifetimes

Generics creates a definition of an abstract-generalized behavior for items like function signatures or structs using generic types instead of concrete types to reduce code duplication.

Traits define behavioral properties in a generic way. Traits are used with generic types to constraint types.

let mut largest = &list[0];

for item in list {

if item > largest { // Order Error: cannot compare all possible T types

largest = item;}}largest}  
fn largest<T>(list: &[T]) -> &T { // finds biggest value inside a list

Lifetimes allow us to borrow values, while enabling the compiler to check validity of references.

## 11.1 Generic Data Types

To define a generic, the type parameter is declared before use it and inside angle brackets <>.

OrderError: We can only use types whose values can be ordered. Use traits to fix this.

#### Performance of code using generics

Your code doesn’t run any slower using generics than concrete types!. Rust accomplishes generics using monomorphization, that means that generic code is turned into specific code by fillling concrete types at compile time.

## 11.2 Traits: Defining shared behavior

Traits are similar to a feature called **Interfaces**. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors.

#### Creating a Trait

pub trait Summary { //creating the Summary trait

fn summarize(&self) -> String;}

Example: You have multiple structs that holds various kinds of text instances: NewsArticle, Tweet and metadata(indicates type of tweet: retweet, reply). We want to make a media aggregator that displays all kinds of text. Hence, we need a to request the summary(trait) by calling summarize method on an instance:

After this method signature, instead of providing an implementation we put a semicolon. Each type implementing this trait must provide its own behavior; the compiler enforces the use of summarize method.

#### Implementing a Trait on a type

pub struct NewsArticle {

pub headline: String,

pub location: String,

pub author: String,

pub content: String,

}

impl Summary for NewsArticle {

fn summarize (&self) -> String {

format!("{}, by {} ({})", self.headline, self.author, self.location)

}

}

To apply the Summary trait for the media aggregator, simply put the **trait** keyword after impl and use for to specify the name of the type we want to implement the trait for.

One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to our crate

#### Default implementations and override

Override is achieved through traits, we can define a default implementation in a trait, and then let the type override or maintain the default behavior.

pub trait Summary {

fn summarize(&self) -> String {

String::from("(Read more...)") //Default implementation

}}

pub struct Tweet {

pub username: String,

pub content: String,

pub reply: bool,

pub retweet: bool,

}

impl Summary for Tweet {

fn summarize(&self) -> String {

format!("{}: {}", self.username, self.content)

}}

#### Traits as parameters

Simply add impl keyword to the parameter:

pub fn notify(item: &impl Summary) {

println!("Breaking news! {}", item.summarize());

}

Or using generics:

pub fn notify<T: Summary + Display>(item: &T) {

println!("Breaking news! {}", item.summarize());

}

Multiple traits can be aggregated used + operator. But if there are many involved, the + makes it unreadable. Clauses(bounds) specify a set of Traits with where keyword.

pub fn notify<T, U>(item: &T, other: &U) -> i32

where T: Summary + Display,

U: Clone + Debug

{

#### Fixing Generic “largest” function with Traits

We ensure that items are comparable by making the generic types implement the PartialOrd trait and items should be able to be copied(only stack located) so generic type should implement Copy trait.

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {

let mut largest = list[0]; //Copy: should be able to copy data

for &item in list {

if item > largest { //PartialOrd: can compare values

largest = item;

}

}

largest

}

fn main() {

let number\_list = vec![34, 50, 25, 100, 65];

let result = largest(&number\_list);

println!("The largest number is {}", result);

let char\_list = vec!['y', 'm', 'a', 'q'];

let result = largest(&char\_list);

println!("The largest char is {}", result);}

Finally, traits can be used to condition the implementation of method.

## 11.3 Validating references with Lifetimes

In the same way that we must annotate types when multiple types are possible, we must annotate lifetimes when the lifetimes of references could live different lifespans. Lifetimes annotations come handy to bound to bound multiple variables to the same lifetime.

#### Borrow checker



}

Compares the scopes of the variables to determine whether all borrows are valid. Here we can see x with a lifetime ‘b and r with a lifetime ‘a. The borrow checker determines that the lifetime of x encloses the lifetime of r.

#### Generic lifetimes annotation

When compiler can’t tell the lifetime inside a function, for example if 2 possible lifetimes are available(if-else could be one case). A lifetime annotation must be provided, describing the relationships of the lifetimes of multiple references to each other.

The syntax uses an apostrophe (‘) and it bounds a lifetime to multiple variable/parameters. For example, a function parameter with lifetime ‘a is applied to all parameters meaning that **all the references in the parameters and the return value must have the same lifetime.**

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {

if x.len() > y.len() {

x

} else {

y

}}

#### Predictable Lifetime by elision rules

Every reference needs a lifetime and you need to specify lifetime parameters for functions or structs that use references. However, because certain reference-lifetime patterns are predictable, Rust programmers programmed patterns into Rust borrow checker called lifetime *elision rules,* they are a set of cases where the compiler can infer the lifetime of the reference.

The elision rules are:

1. Each parameter that is a reference gets its own lifetime parameter. In other words, a function with one parameter gets one lifetime parameter: fn foo<'a>(x: &'a i32); a function with two parameters gets two separate lifetime parameters: fn foo<'a, 'b>(x: &'a i32, y: &'b i32); and so on.
2. if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: fn foo<'a>(x: &'a i32) -> &'a i32.
3. if there are multiple input lifetime parameters, but one of them is &self or &mut self because this is a method, the lifetime of self is assigned to all output lifetime parameters. This third rule makes methods much nicer to read and write because fewer symbols are necessary.

#### The static lifetime

‘static is a reference that can live for the entire duration of the program.

# 12 Writing automated test

## 12.1 How to write tests

Test functions typically perform 3 actions:

* Set up any needed data or state.
* Run the code you want to test.
* Assert the results are what you expect.

Rust offers test and should\_panic attributes and macros to achieve these actions.

#### Creating test functions

To change a function into a test function, add #[test] before fn. Test can be run with “cargo test” command for which Rust builds a test runner **special binary** that contains and reports only the test annotated functions.

cargo new adder –lib -> Creates a new library for test function

cargo test -> runs test library.

|  |  |
| --- | --- |
| Function | Description |
| Assert\_eq!(a,b) | Passes test only if a == b |
| assert\_ne!(a,b) | Passes test only if a != b |
| Assert!(a) | Passes only if a == true |
| Use super::\*; | Brings anything written in the outer scope to the test module, to be able to use any declared function of src. |
| #[test] | Converts a function into a test function |
| #[cfg(test)] | Tells Rust to compile the module only when cargo test command is run. |

## 12.2 Controlling how test are run

You can select a single test to run or select multiple by pattern matching.

## 12.3 Test organization

Unit tests: built within the module, test each unit of code in isolation. -> The convention is to create a module named test INSIDE THE .rs file! and annotate it with cfg(test) in each file that contain test functions.

// module.rs

fn run(){…}

…

#[cfg(test)]

mod test {

use super::\*;

#[test]

fn test\_run() {…}

Integration tests: Integration tests are entirely external to your library. Only calls functions that are part of the public API. They test coherency between parts of the library. -> Create a tests directory at the top level of the project, next to src. Thes only require #[test].

Appendix A: In deep

A1. Enumerators

Result Enumerator: is a type that represents either success ([Ok](https://doc.rust-lang.org/std/result/enum.Result.html#variant.Ok)) or failure ([Err](https://doc.rust-lang.org/std/result/enum.Result.html#variant.Err)). It helps to propagate errors.

pub enum Result<T, E> {

Ok(T),

Err(E),

}

Appendix C: Compiler

C1. Immutability:

The compiler guarantees that when you state immutable variables, the value really won’t change. This is done by getting compiler errors.

Note that mutating an instance in place is faster than copying and returning newly allocated instances.