Performance

Security

User-friendly

Robust ecosystem

* Memory Management

Power of C but with ease of syntax

The History of Rust

2006
Origin
Rust begins as a personal project of Graydon Hoare

2009
Sponsorship
Mozilla officially begins *supporting Rust development*

2015
Rust 1.0
The first stable release of Rust occurs

2021
Foundation
The Rust Foundation is established

2021
Rust 2021
In October Rust edition 2021 is released

2022
Growth
Rust is used in Linux kernel development

Why Use Rust?

Performance

Memory Management

Memory safety without sacrificing performance

Ownership

You can experience the “if it compiles, it works” phenomenon

Excellent and growing ecosystem

Pleasing syntax and powerful language features

rust → cargo → package manager & build system

cargo new project_name

cargo.toml

toml → Tom's Obvious minimal Language

name = John
age = 30

Compile

```
rustc filename.rs
```

```
./filename
```

If you want to make it mutable → mut

Variables are immutable by default

Casings: → one two three four

Pascal Casing → OneTwoThreefour

Camel casing → oneTwoThreefour

Snake casing → one_two_three_four

→ Rust uses snake

```
// Variables
```

let v/s const

```
let my_number = 1;
```

```
let mut my_other_number = 2;
```

- **Defining a mutable variable**

```
const IMPORTANT_NUMBER: i32 = 42;
```

- **Defining a constant. A constant is always immutable and valid throughout the life of your program.**

```
let my_number = 2;

// "my_number" now points to the number "2"
```

- **Shadowing a variable is possible. This is how to perform a transformation on a value where the variable remains immutable.**

Comments

Simple

Documentation

```
// Comments
```

Normal Comment

```
let my_number = 2; // Another comment
```

- Comments can also be placed at the end of a line of code

Documentation Comment

```
/// Subtracts "y" from "x"

pub fn subtract(x: i32, y: i32) -> i32
    {  x - y
}
```

- Documentation comments are created with three forward slashes. These are used when creating external crates. Documentation is generated from comments like these.

crate → compilation unit that can hold multiple modules.

↳ directory - name

#[crate =]

To create doc

cargo doc

Primitive Data Types in Rust

Data Type Categories

Scalar

These are types that represent a single value

Compound

Types that represent multiple values

Scalar Types

Integer

Floating Point

Boolean

Character

```
// The Integer Type

let my_number: i32 = 1;

let my_8_bit_int: i8 = -128;

let other_int: u8 = 128;
```

+
-

positive

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

architecture of OS

```
// More Primitive Scalar Types
```

```
let my_floating_point: f64 = 2.5;

let other_float: f32 = 2.0;
```

- **Defining floating point values**

```
let this_is_true: bool = true;

let this_is_false = false;
```

- **Defining booleans**

```
let my_char: char = 'A';
```

- **Defining a single character literal**

Compound Types

let my_tuple = ('A', 1, 2, 'B');

Tuples

varying length

indexed access

0 based

Destructuring

Arrays

fixed length

indexed-access

let (_, second_el, _, _) = my_tuple

```
// Primitive Compound Types
```

```
let my_tuple = ('A', 1, 1.2);
```

my_tuple.0

```
let my_arr = [1, 2, 3];
```

- **Defining an array**

```
let my_arr_typed: [i32; 3] = [1, 2, 3];
```

- **Defining an explicitly typed array**

Creating Custom Types

Custom Types

Structs

A combination of different data types which create a custom type

Enums

A way to define a custom type by explicitly defining its states

```
// Example struct


struct Coffee {

  id: i64,

  name: String

}
```

```
let coffee = Coffee {

  id: 123456,

  name: String::new("Latte")

}
```

- **Structs are defined by using the "struct" keyword alongside a name**
- **Structs are your means of combining different data types together underneath the umbrella of a custom type**
- **Instantiating a struct looks like this**

```
// Example enum

enum CoffeeType

  {  HOT,

  ICED

}

// Example usage

let my_coffee = Coffee

  {  id: 123456,

  coffee_type: CoffeeType::HOT,

  name: String::new("Latte")
}
```

t **Using an enum looks like this**

```
// Including data with enums

enum CoffeeType

  {  HOT(f32),

  ICED(f32)

}

// Example usage

let my_coffee = Coffee

  {  id: 123456,

  coffee_type: CoffeeType::HOT(102.5),

  name: String::new("Latte")
}
```

t **Example of an enum with data attached. Here we have the temperature attached to it.**

```
let my-tuple = (1, 2, 3, 4);
```

one

two

```
let one = my-tuple.0
let two = my-tuple.1
let three = my-tuple.2
let four = "          ")
```

```
let (one, two, three, four) = my-tuple;
```

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

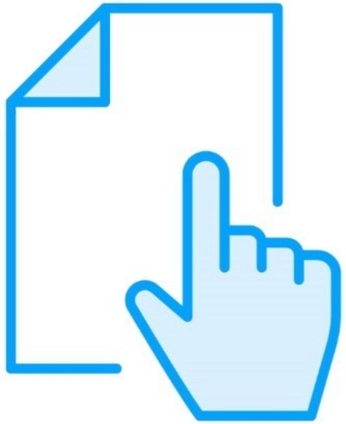
Represents a value of Type T

Option

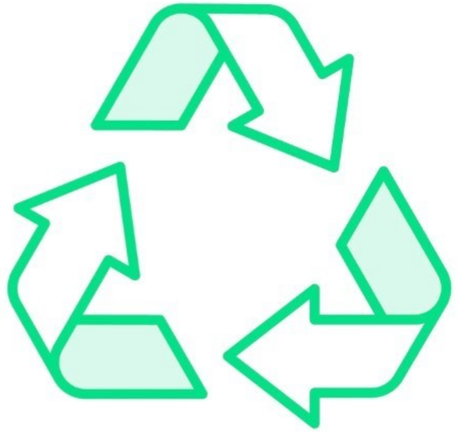
Represents absence of value

Approaches to Memory Management

OWNERSHIP & REFERENCES



→ Rust falls in this category



Manual Memory Management

In this category, programming languages require the developer to manage memory.

Garbage Collected

Languages in this category use a garbage collector – a program that is responsible for managing memory

The Rule of Three

1

Every value is owned

2

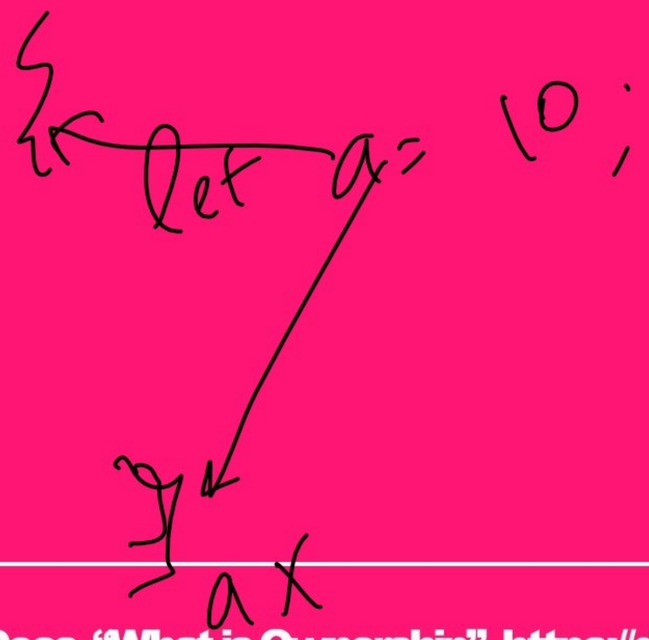
One owner at a time

3

When the owner leaves scope, what is owned gets dropped!

Scope

Scope is the range within a program for which an item (a value) isvalid.

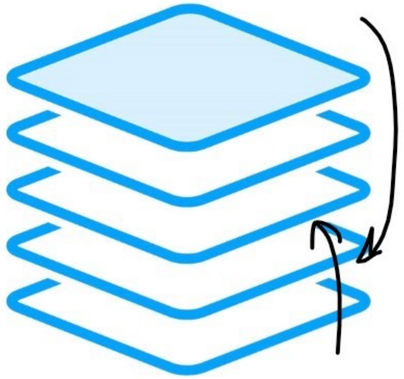


Citation: Rust Docs, "What is Ownership", https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

The Stack and the Heap

Push

Pop



The Stack

Memory that is immediately available to your program and that is ordered (last-in first-out)

The Heap

A chunk of memory that is broadly available to your program. Your program must request what it needs

The Stack vs. the Heap

Stack	VS	Heap
Fast – the memory allocator does not need to request access		Slower – must request access and follow a pointer
Contains ordered values - LIFO		Values are not necessarily ordered
Think "compile-time" - everything on the stack has a known, fixed-size		Memory is allocated at runtime

```
// Example of simple stack-based scope

{

  let my_number = 1;

}

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

- Let's imagine this code resides within a "main" function
- Defining a scope using curly braces.
- Within this scope, "my_number" owns this value.
- This will not compile! The value pointed to by the "my_number" variable is no longer in scope. The value has been "dropped".

```
// Ownership on the heap

{

  let my_string = String::from("Hello");

}
```

- Allocate memory on the heap for a String
- The "my_string" variable is no longer valid – behind-the-scenes, Rust calls the "drop" function which frees up this memory!
- Easy - no need to manually free the memory

Stack

1. Function Call Frames.
   ↳ when a function is called a new stack frame is created.

2. Local Variable
   integer, boolean

Automatic Deallocation

Heap

1. Dynamic Allocation.
   → where the size of data is not known at compile time.

2. Owned Data
   Data whose size can change or not known until runtime String Vec

3. Ownership & Borrowing.

Box<T> Manual Deallocation
drop

```
// Moves/Copies - transferring ownership

// On the stack

{

  let a = 1;

  let b = a;


}

// On the heap

{

  let str_a = String::from("Hello");

  let str_b = str_a;

}
```

- **Variables "a" and "b" can both be used as these are simple values that are stored on the stack. Here, "b" is a copy (think shallow-copy) of "a".**

- **Here the "str_a" variable has stack data that points to data on the heap. The stack data is copied to "str_b". The heap data is left untouched. This is known as a "move".**

- **Use of the "str_a" variable is now invalid.**

```
// Cloning - making deep copies

{

  let str_a = String::from("Hello");

  let str_b = str_a.clone();

}
```

- Sometimes we need to make a copy of data on the heap.
- We can do this on data structures that implement the "Clone" trait.
- Here we make a deep copy of a string.
- The variables "str_a" and "str_b" are both valid and both reference deeply-cloned data on the heap.

References and Slices

Categories

References

Immutable or mutable links back to some piece of data

Slices

A special reference that points to a subset of data

Rules with References

1 → mutable ref

N → Immutable ref

x → y

One mutable reference or any amount of immutable references

References HAVE to be valid

```
// References and borrowing

fn concat(a: &mut String)

  {  a.push_str("bar")

}

let foo = String::from("foo");

concat(&foo); // We can't do this!

let mut other_foo = String::from("foo");

concat(&mut other_foo);
```

&mut

- **Function that takes a String reference**
- **Immutable reference**
- **Mutable reference**

allows modification of data!

allows exclusive access

Performance

large datasets

it avoid allocating de

```
// Slices - strings and arrays


let foobar = String::from("foobar");

let foo = &foobar[0..4];

let bar = &foobar[3..]


let my_arr = [ 1, 2, 3 ];

let one = &my_arr[..1];

let my_mut_arr = [ 1, 2, 3];

let my_mut_slice = &mut my_mut_arr[..1];

my_mut_slice[0] = 2;

// my_mut_arr now looks like: [2, 2, 3]
```

&foobar[0..4]
// slice of index

0 → 4-1

- **String variable**
- **A string slice that references the string "foo"**
- **A string slice that references the string "bar"**
- **Arrays have slices too!**
- **Mutable array slice**
- **Mutate by an index (unsafe - this is just an example)**

0 1 2 3 4 5

foobar

1 2 3 4 5 6

0 : 5

5 - 1

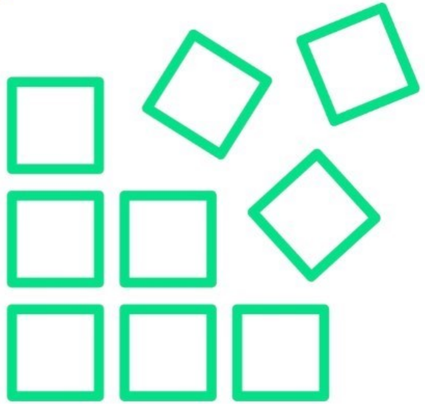
Hello World

0 1 2 3 4 5 6 7 8 9

0 = 4

Control Flow

Rust is statement based



Expressions

A combination of values, variables, operations, etc. that the compiler evaluates into a new value

Statements

A standalone unit of code that does not return a value. Think "structure" here.

Rust is expression-based

```
if my_num >= 5 {
    println!("The number is greater than or equal to 5!");
} else if my_num > 2 {
    println!("The number is greater than 3 but less than 5!");
} else {
    println!("Now we are running this code!");
}
```

```
let should_use_ten = true;

let my_num = if should_use_ten { 10 } else { 0 };
```

```
let my_num=0;
if(true){
my_num=10
}
else
{
}
```

loop

while

for

```
// Basic infinite loops
loop {
    println!("We are looping!");
}


// Returning values using loop
let mut my_num = 0;
let final_num = loop
    {  my_num += 5;

    if my_num > 10
        {  break my_num +
        1;
    }
}
```

```
let mut my_num = 0;

while my_num < 10
  {  my_num += 1;
 // This code will run until “my_num” is less than 10
}
```

for element in collection{

```
let my_arr = [1, 2, 3, 4, 5];

for num in my_arr {
    // Here we have access to each element in the array
}

// Mutable looping
let mut my_vec = vec![1, 2, 3, 4, 5];

for num in &mut my_vec {
    *num += 1;
}
```

}

Pattern Matching

"…an extremely powerful control flow construct …that allows you to compare a value against a series of patterns and then execute code based on which pattern matches."

Citation: Rust Docs, "The match Control Flow Construct", https://doc.rust-lang.org/book/ch06-02-match.html

```
enum MessageState
  {  Pending,
  Sending,
  Received
}

let msg_state = MessageState::Pending;

match msg_state {
  MessageState::Pending => println!("Message Pending!"),
  MessageState::Sending => println!("Message Sending!"),
  MessageState::Received => println!("Message Received")
}
```

```
let msg_state = MessageState::Pending;

let status_code = match msg_state
  {  MessageState::Pending => 1,
  MessageState::Sending => 2,
  MessageState::Received => 3
}
```

```
enum MessageState
  {  Pending(1),
  Sending(2),
  Received(3)
}

let msg_state = MessageState::Received;

match msg_state {
  MessageState::Pending(status_code) => println!("Msg status: {}", status_code),
  _ => println!("Message is no longer pending")
}
```

Option

None → indicates failure with No Value

Some(T) → a value of T

Result

Ok(T) → operation succeeded with a value T

Err(E) → operation failed with a error E

Result

Ok(T)

Err(E)

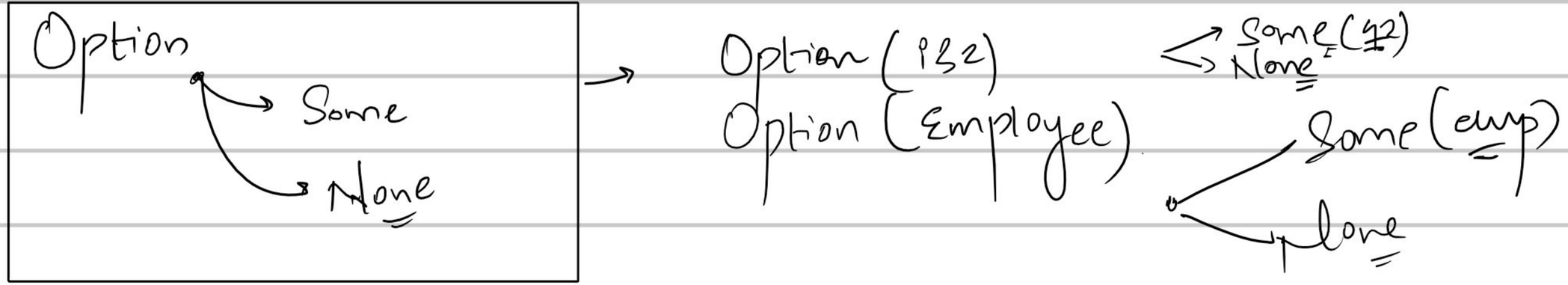
what type will be returned in case of success

what type will be returned in case of error

Result<i32, &str>

Ok(222)

Err("")




```
enum Color {
    GREEN,
    RED,
    BLUE
}
```

```
let some_color = Color::RED;
match some_color {

    Color::RED => ——
    Color::GREEN ——

}
```

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

```
let my_opt : Option<i32> = Some(123);
                           None

match my_opt {
    Some(value) => ____,
    None        => ____
}
```

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

```
let my_result : Result<f32, &str> = Ok(102.5)
                                    Err("Coffee shop is closed");
match my_result {
    Ok(value)  => ____,
    Err(error) => ____
}
```

Ok

Err

```
enum MessageState
  {  Pending(1),
  Sending(2),
  Received(3)
}

let msg_state = MessageState::Received;

if let MessageState::Pending(status_code)  = msg_state {
  println!("Msg status: {}", status_code),
}

// Can use an optional "else" block afterwards!
```

Pattern matching is powerful!

! - NOT

Reverses the logical state of its operand. True becomes false. False becomes true

&& - AND

If both operands are true, returns true. If any operand is false, returns false.

|| - OR

If any operand is true it returns true, otherwise returns false

```
let this_is_true = true;
let this_is_false = false;

if !this_is_false {
  println!("This code will run - false becomes true!");
}

if this_is_true || this_is_false {
  println!("This code will run - one of these operands is true!");
}

if this_is_true && this_is_false {
  println!("This code will NOT run - one of these operands is false!");
}
```

!=

>

<

==

>=

<=

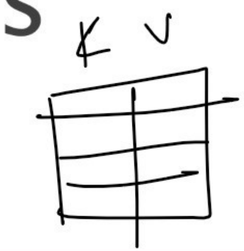
```
let my_num = 5;
if my_num != 3 {
  println!("This code will run - 'my_num' does not equal 3!");
}

if my_num == 5 {
  println!("This code will run - 'my_num' equals 5!");
}

if my_num > 6 {
  // This block won't run
} else if my_num >= 6 {
  // This block won't run
} else if my_num < 5 {
  // This block won't run
} else if my_num <= 5 {
  // This block WILL run because 'my_num' is equal to 5
}
```

Collections in Rust 2021

Iterators

↳ responsible for creating a sequence of values & allow us to iterate over each item in sequence.

Collection Categories



Array, Vector,

Sequences

Maps

Sets

Misc.

Sequence Types

Vec

A growable sequence of elements - contiguous in memory

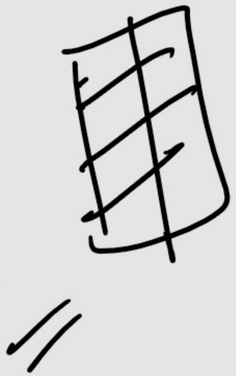
VecDeque

A double-ended queue – not always contiguous in memory

LinkedList

A doubly-linked list that contains a series of nodes

Map Types



HashMap

A key-value store

BTreeMap

A map optimized for search

Set Types

HashSet

A set containing no duplicates implemented as a HashMap

BTreeSet

An ordered set – optimized for search

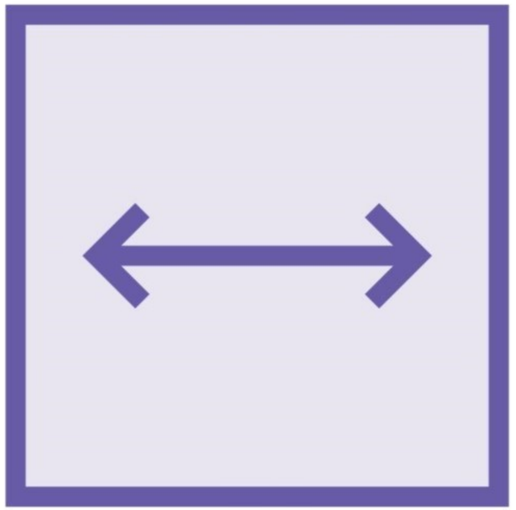
Miscellaneous

Binary Heap

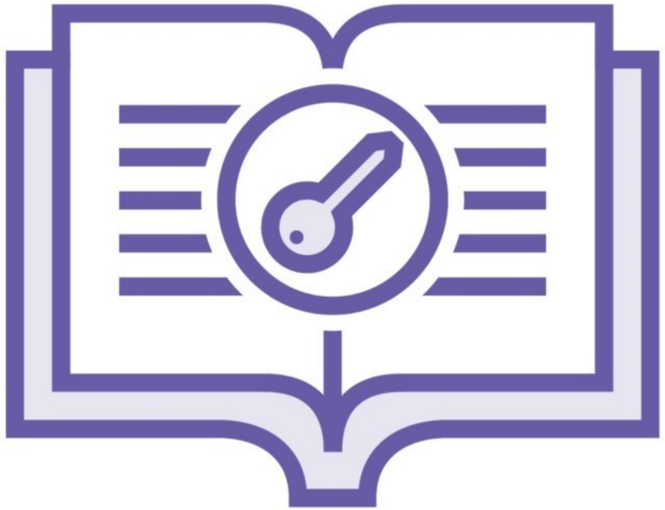
A priority queue implementation

Practical Collections

Most Useful Collections



Vec
Used for containing any growable sequence of data



HashMap
Used as a “dictionary” or key-value store

Strings
Technically a ”Vec” – a collection of bytes

```
// Simple examples of vectors/strings

let mut my_vec = vec![2, 4, 6];

my_vec.push(8);

let hello = String::from("Hello");
```

- Initializing a Vec using the "vec!" macro
- Adding a value to the end of a vector
- Initializing a String – a high-level wrapper around a vector of bytes

```
// Simple examples of the HashMap

let mut coffee_ratings = HashMap::new();

coffee_ratings.insert("Latte",10);

for (name, rating) in &coffee_ratings {
  println!("{name} rating: {rating}");
}
```

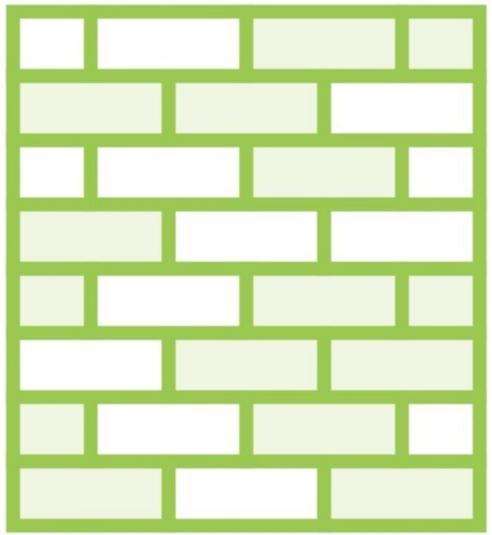
std::collections::HashMap

- **Initialize a hashmap**
- **Inserting a key-value pair**
- **Print each key-value pair to the console**

Tuples, Arrays and Slices

Arrays

[1,2,3]



Fixed-size

Unlike vectors, arrays have a fixed size. You cannot grow an array.

Contiguous in memory

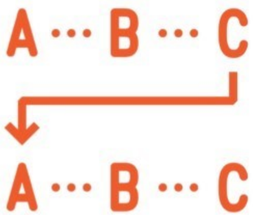
The bytes that comprise an array are laid out in a contiguous block of memory

Tuples

→ Heterogeneous

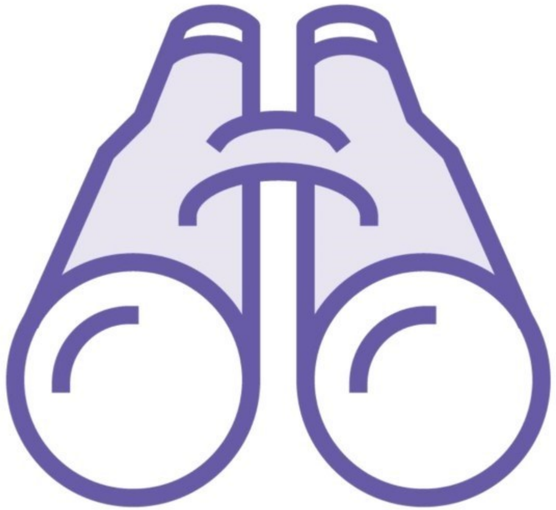
Finite – different lengths means different types altogether

Sequence of elements



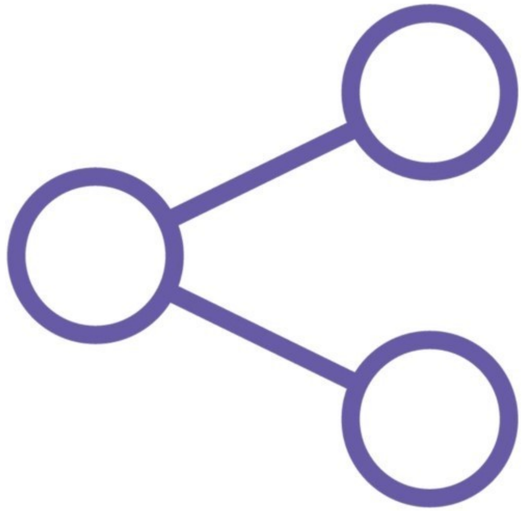
Elements can be of different types

Slices



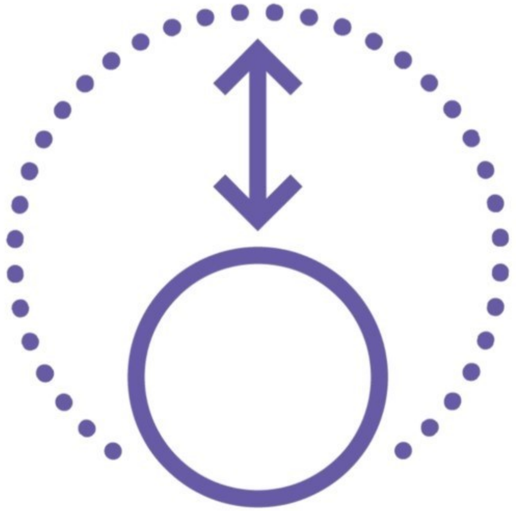
Views

Slices are views into a block of memory



Mutable or Shared

Mutate or provide a simple reference to a block of memory



Dynamically Sized

The size of a slice is known only at run-time

```
// Tuples, Arrays and Slices

let tuple = (1, 2, “Hello World”);

let my_array: [u8; 4] = [1, 2, 3, 4];

let my_slice = &my_array[1..3];

// my_slice looks like: [2, 3]
```

- **Define a tuple**
- **Defining an array given a type and size**
- **Take a slice of “my_array” which contains the second and third elements given an inclusive starting index and an exclusive ending index**

Working With Sequences

Vec
VecDeque
LinkedList

Vec

A contiguous growable array type, written as Vec<T>…

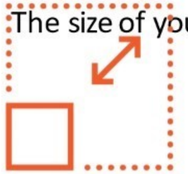
Citation: Rust Documentation, Module std::vec, https://doc.rust-lang.org/stable/std/vec/

Vec is short for “Vector”

Why Use Vec?

You want an ordered sequence of items

You want a stack



The size of your collection will grow dynamically You want to access items by index

You want to allocate memory for your sequence on the heap

Heap

The most robust anduseful sequence type!

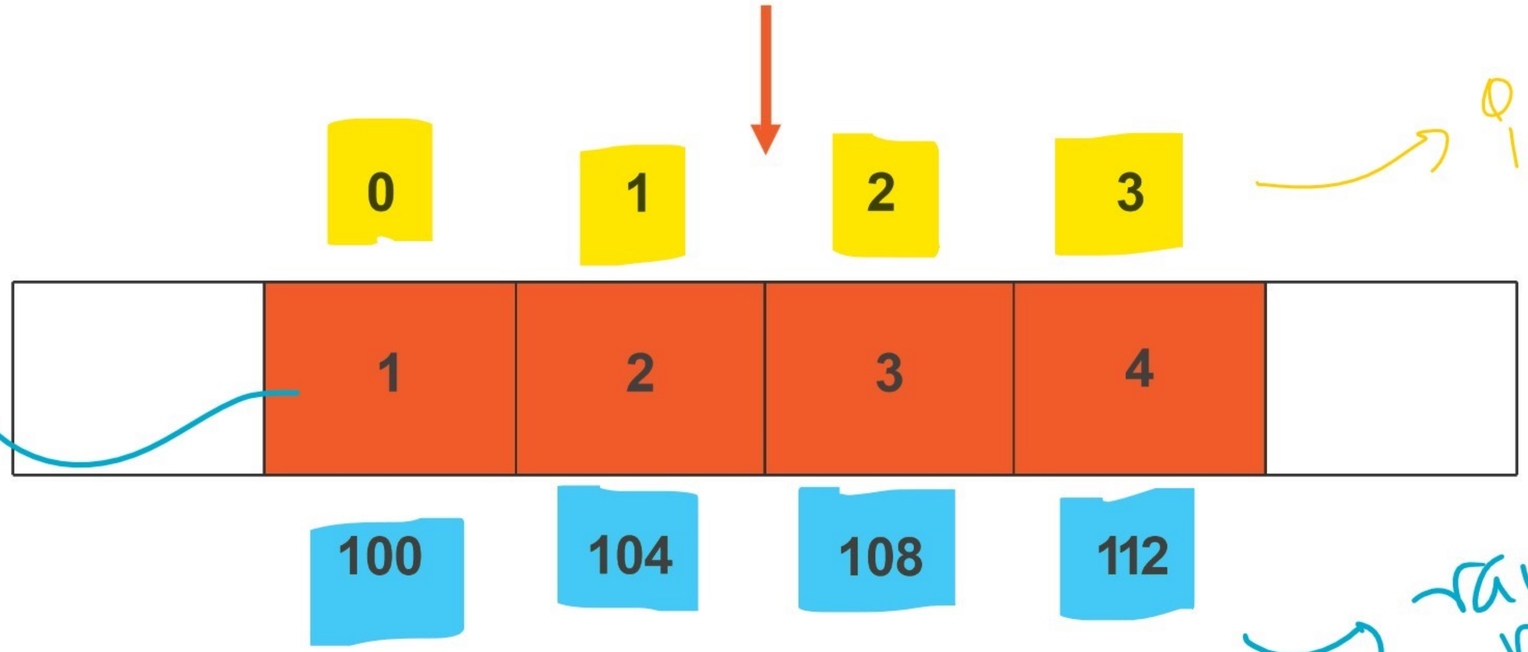
Contiguous Heap Allocation

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

Contiguous Heap Allocation

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

elements of vector



index numbers

random memory address

Useful Vec Functions

Practical Focus

Initialization

General Use

```
let my_vec = vec![0, 1, 2, 3, 4];


let my_other_vec = vec![0; 6];

// [0, 0, 0, 0, 0, 0]
```

Initialization using the “vec!” macro

A very useful and expedient way of initializing a vector in Rust is via use of the “vec!” macro. The first example shows the most common use-case.

The second example shows quickly initializing a vector given a size and uniform value.

```
let from_vec = Vec::from([0, 1, 2]);
```

```
let mut new_vec = Vec::new();

new_vec.push(1);


let cap_vec = Vec::with_capacity(5);
```

Initialization using “new”, “from”, and “with_capacity”

A vector can also be initialized using the ”new”, “from” and ”with_capacity” functions.
A vector initialized via “new” is not allocated memory until elements are pushed onto it.
The “with_capacity” function is useful for initializing an empty vector which is pre-allocated. The “from” function is useful for initializing a vector from another sequence – this time an array.

General Utility

```
// Indexed access

let mut my_vec = vec![2, 4, 6];

my_vec[0] = 0;

let my_num = my_vec[2];

my_vec.get(0); // Returns an Option type

my_vec.get_mut(0) // Can mutate
```

- **Setting a value at a given index**
- **Retrieving a value at a given index**
- **Safer way of retrieving a value at a given index**
- **”get_mut” can be used to mutate the value in the containing vector**

```
// Stack-based approach

let mut my_vec = vec![2, 4, 6];

my_vec.push(8);

let maybe_num = my_vec.pop();

// .pop returns an Option type

// The length of my_vec is now 2
```

- **Adding a value to the end of the stack**
- **Popping a value off of the top of the stack**

```rust
// General utility functions

let mut my_vec = vec![2, 4, 6];


my_vec.len(); // 3

my_vec.append(&mut vec![8, 10]);

my_vec.clear();


my_vec.drain(1..);


my_vec.insert(1, 10);  // [2, 10, 4, 6]

my_vec.remove(1); //  [2, 6]
```

- Combining two vectors
- **Getting the length of a vector**
- Removing all values from a vector
- Removes range of elements from a vector - returns an iterator
- Inserts a value at a given index – shifting all other elements to the right
- Removes a value at a given index – shifting other elements to the left

```rust
// General utility functions

let mut my_vec = vec![2, 4, 6];

my_vec.retain(|&n| n > 2); // [4, 6]


my_vec.retain_mut(|n| if *n > 2 {

  *n += 2;

  true

} else {

  false

}); // my_vec => [6, 8]


my_vec.truncate(1); // [2]
```

- **Filters out values from a vector given a predicate**
- **Allows mutation of retained values**
- **Removes all values after the given length**

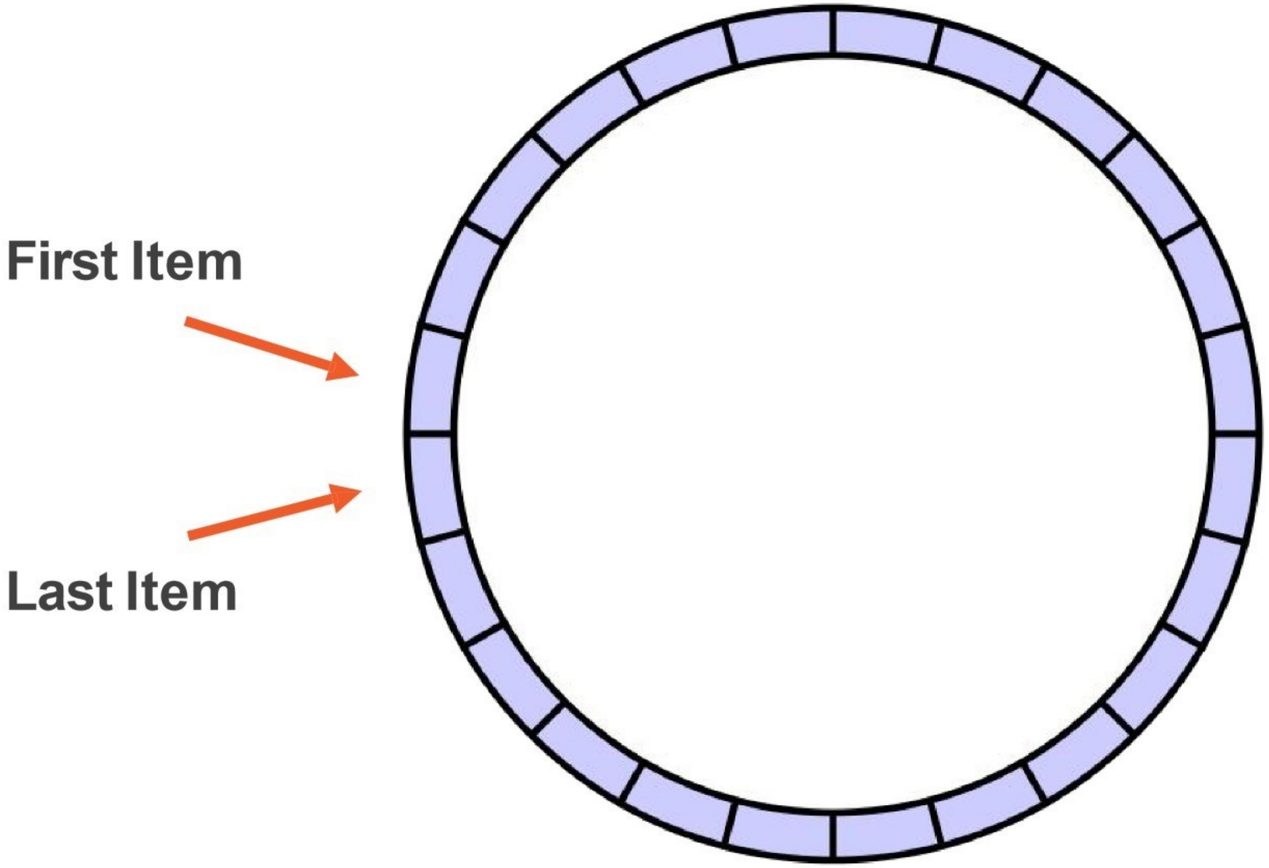
Introducing The VecDeque Type

VecDeque

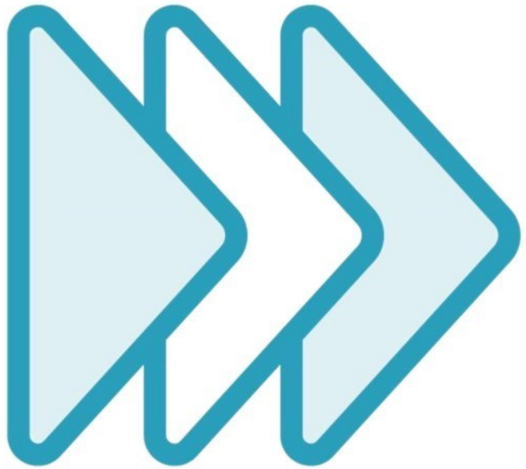
A double-ended queue implemented with a growable ring buffer.

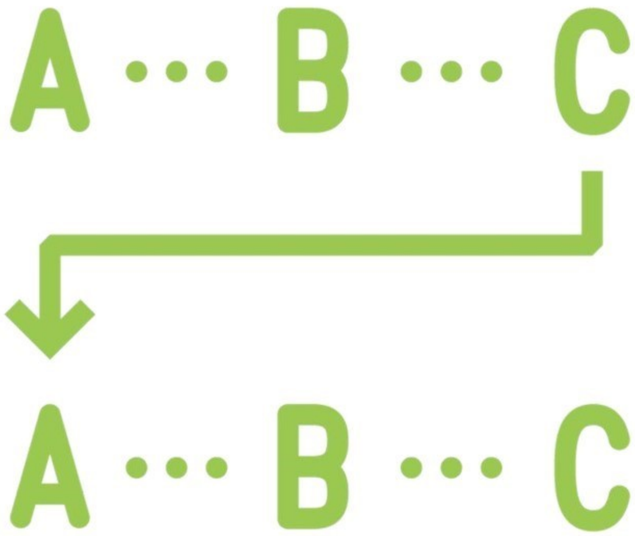
Citation: Rust Documentation, std::collections::VecDeque, https://doc.rust-lang.org/

Ring Buffer



VecDeque





Insertion Efficiency

The double-ended queue is a good choice if you need efficient insertion/removal at both ends

Queue Functionality

If your problem domain needs a queue – this is the Rust data structure you want

A VecDeque is not always contiguous in memory.

Useful VecDeque Functions

```
let new_deque: VecDeque<u32> = VecDeque::new();

let cap_deque: VecDeque<u32> = VecDeque::with_capacity(5);

let from_deque: VecDeque<u32> = VecDeque::from([0, 1, 2]);
```

Initialization using “new”, “from”, and “with_capacity”

A VecDeque can be initialized using the ”new”, “from” and ”with_capacity” functions.

```
let deque: VecDeque<u32> = VecDeque::new();
```

```
deque.push_back(1);
deque.push_front(2);
deque.pop_back();
deque.pop_front();
deque.contains(1);
```

- Insert elements at the front and back of the queue
- Remove elements from the back and front of the queue
- Check to see if queue contains a given value

```
front();
front_mut();
back();
back_mut();
```

- Get a readable or mutable reference to the value at the front or back of the queue

General utility functions from “Vec” still apply!

Introducing The LinkedList Type

LinkedList

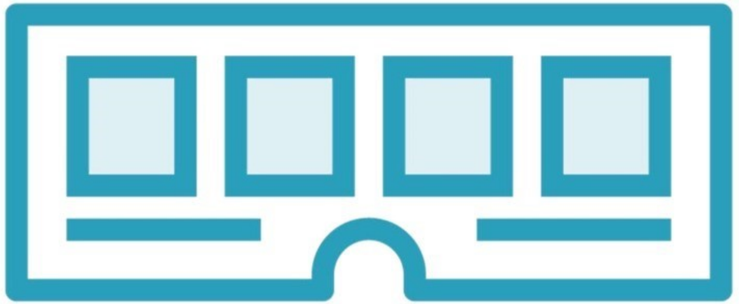
A doubly-linked list with owned nodes.

NOTE: It is almost always better to use `Vec` or `VecDeque` because array-based containers are generally faster, more memory efficient, and make better use of CPU cache.

Citation: Rust Documentation, std::collections::LinkedList, https://doc.rust-lang.org/

Only use the LinkedList if you are absolutely sure you really need it!

LinkedList



Avoiding Amortization

The doubly-linked list is good for when you want to avoid the performance costs of reallocations

Efficient split/append

According to the Rust docs, the LinkedList data structure is efficient at splitting/appending.

Useful LinkedList Functions

```
let new_list: LinkedList<u32> = LinkedList::new();

let from_list: LinkedList<u32> = LinkedList::from([0, 1, 2]);
```

Initialization using “new” and “from”

A LinkedList can be initialized using the ”new” and “from” functions.

```
let list: LinkedList<u32> = LinkedList::new();
```

```
list.push_back(1);
list.push_front(2);
list.pop_back();
list.pop_front();
list.contains(1);
```

- Insert elements at the front and back of the linked list
- Remove elements from the back and front of the linked list
- Check to see if the list contains a given value

```
front();
front_mut();
back();
back_mut();
```

- Get a readable or mutable reference to the value at the front or back of the linked list

Most utility functions from Vec do NOT apply

Using Maps And Sets

HashMap / BTreeMap

The HashMap is an abstract data type that maps keys to values.

The Rust BTreeMap is an ordered HashMap that is based on a B-tree.

Citation: Wikipedia, Hash Table, https://en.wikipedia.org/wiki/Hash_table

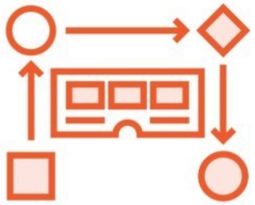
Citation: Rust Documentation, std::collections::BTreeMap, https://doc.rust-lang.org

Alongside the Vec type, HashMap is the most useful collection type

Why Use HashMap?

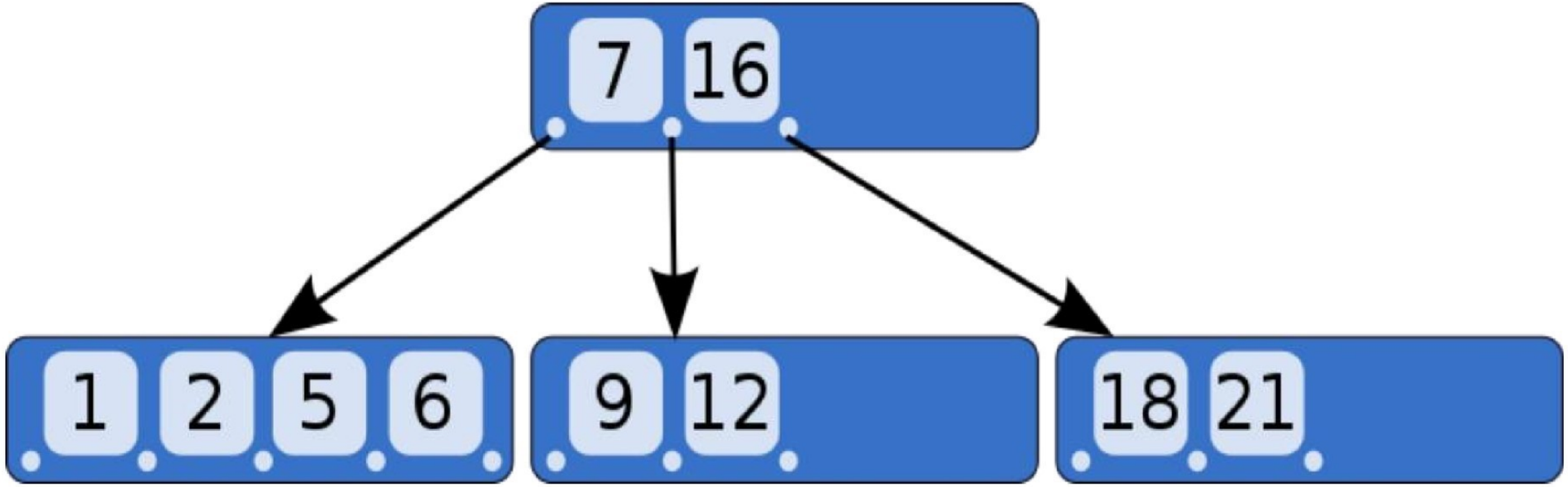
You want to store data as key-value pairs

You want fast access of elements



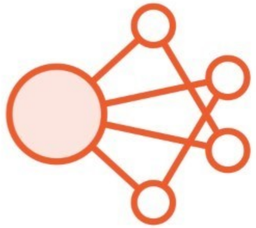
You want to create an in-memory cache

B-Tree

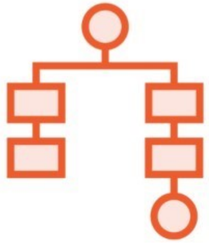


Why Use BTreeMap?

You want your map optimized for search



You want your map sorted



Ordering is key

Choose BTreeMap if you careabout how your map is ordered and you want to search throughit!

Useful Map Functions

```
let mut new_map = HashMap::new();

let mut from_map = HashMap::from([(“Latte”, 3.50)]);

let mut cap_map: HashMap<&str, i32>
 = HashMap::with_capacity(5);
```

Initialization using “new”,“from”,and “with_capacity”

A HashMap can be initialized using the ”new”, “from” and ”with_capacity” functions.

Alternatively, the “with_hasher” and “with_capacity_and_hasher” functions can be used if you want to specify your own hasher. This hasher must implement the BuildHasher trait.

```
let mut new_map = BTreeMap::new();

let mut from_map = BTreeMap::from([(“Latte”, 3.50)]);
```

Initialization using “new” and “from”

A BTreeMap can be initialized using the ”new” and “from” functions.

Note that “with_capacity” does not exist for BTreeMaps!

```
// HashMap Common Operations

let mut my_map = HashMap::new();

my_map.clear();
my_map.len();

my_map.insert("Latte", 3.50);

my_map.get("Latte"); // An Option type
my_map.get_mut("Latte"); // Can mutate
my_map.remove("Latte");
```

- Remove all key-value pairs from the map
- Get the count of key-value pairs in the map
- Setting a key-value pair – note that if the key already exists, the value is updated.
- Safe way of retrieving a value
- "get_mut" can be used to mutate the value in the containing HashMap
- Remove a key from the HashMap

```
// HashMap Common Operations
```

```
my_map.keys();
```

- Returns an iterator that loops over all keys in the map randomly

```
my_map.values();
my_map.values_mut();
```

- Returns an iterator that loops over all values in the map randomly

```
my_map.into_keys().collect();
my_map.into_values().collect();
```

- Similar to the above functions but these cause the map to be unusable.

```
my_map.get_key_value();
```

- Useful function for extracting a key-value pair from a map by a given key

```rust
// BTreeMap Common Operations

let mut tree_map = BTreeMap::new();


tree_map.first_entry();

tree_map.last_entry();


tree_map.pop_first();

tree_map.pop_last();


let mut other_map = BTreeMap::new();

tree_map.append(&mut other_map);
```

- **Since a BTreeMap is an ordered map, there are utility functions for retrieving the first and last entries**

- **You can retrieve and remove the first and/or last values from a BTreeMap**

- **You can combine two BTreeMaps. The entries will be ordered and overlapping keys will be overwritten.**

General utility functionsfrom “HashMap” still apply!

Introducing The Set Types

HashSet / BTreeSet

The HashSet is a HashMap where the value is a “unit” type.

The Rust BTreeSet is an ordered set that is based on a B-tree.

Citation: Rust Documentation, std::collections::HashSet, https://doc.rust-lang.org

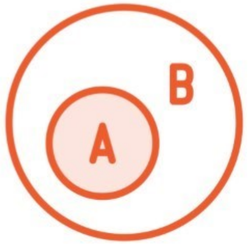
Citation: Rust Documentation, std::collections::BTreeSet, https://doc.rust-lang.org

Fit very specific usecases

Why Use HashSet?

You don’t really care about storing values – just keys

You are concerned about duplicates



Your problem domain requires set operations

Use the BTreeSet when you want an ordered HashSet

Useful Set Functions

```
let mut new_set = HashSet::new();


let mut from_set =
HashSet::from([“Latte”, “Coffee”]);


let mut cap_set: HashSet<i32> =
 HashSet::with_capacity(50);
```

Initialization using “new”, “from”, and “with_capacity”

A HashSet can be initialized using the ”new”, “from” and ”with_capacity” functions.

Alternatively, the “with_hasher” and “with_capacity_and_hasher” functions can be used if you want to specify your own hasher. This hasher must implement the BuildHasher trait.

```
let mut new_set = BTreeSet::new();
```

```
let mut from_set =
BTreeSet::from([“Latte”, “Coffee”]);
```

Initialization using “new” and “from”

A BTreeSet can be initialized using the ”new” and “from” functions.

Note that, much like the BTreeMap, “with_capacity” does not exist for the BTreeSet.

```
// HashSet Common Operations

let mut my_set = HashSet::new();

my_set.clear();
my_set.len();

my_set.insert("Latte");
my_set.remove("Latte");

my_set.contains("Latte");
```

- Shares many common functions with the HashMap

- Remove all values from the set
- Get the count of values in the set

- Add a value – note that if the value already exists, the function returns "false"
- Remove a value

- Returns a boolean denoting whether or not the value is in the set

```rust
// HashSet Common Operations

let mut set1 = HashSet::new();

let mut set2 = HashSet::new();


set1.difference(&set2);

set1.intersection(&set2);


set1.is_subset(&set2);

set1.is_superset(&set2);

set1.is_disjoint(&set2);

set1.symmetric_difference(&set2);

set1.union(&set2);
```

- **HashSets have the common operations for sets across all programming languages, many of which return an iterator.**

- Returns an iterator for the values that arein set1 but not in set2
- Returns an iterator for the values that areonly in both set1 and set2

- Returns boolean values that show whether or not two sets have common values (subset, superset) or not (disjoint).

- Which values are unique to eachset

- All values from both sets – no duplicates

```rust
// BTreeSet Common Operations

let mut tree_set = BTreeSet::new();


tree_set.first();

tree_set.last();


tree_set.pop_first();

tree_set.pop_last();


let mut other_set = BTreeSet::new();

tree_set.append(&mut other_set);
```

- **The BTreeSet shares common operations with the BTreeMap and HashSet.**

- **Like the BTreeMap, you can retrieve and remove the first and/or last values from a BTreeSet**

- **You can combine two BTreeSets. The resulting BTreeSet will be ordered without duplicates.**

Working With Strings

String

A String is stored as a vector of bytes, but guaranteed to always be a valid UTF-8 sequence.

Citation: Rust Documentation, Strings, https://doc.rust-lang.org/rust-by-example/std/str.html

Two Types

Primitive “&str”

The “&str” type is a slice

Wrapper ”String”

Higher level of abstraction – “heap allocated, growable, not null terminated”

String andstr

```
let primitive_str: &str = “Hello World!”;

let mut my_dynamic_string = String::new();

let mut hello_world_chars: Vec<char> = primitive_str.chars().collect();

for c in hello_world_chars {
  my_dynamic_string.push(c);
}

let just_hello: Option<&str> = my_dynamic_string.strip_suffix(“ World!”);
```

Useful String Functions

```
let mut my_str = String::from("Hello");
```

```
let my_bytes = my_str.as_bytes();
let str_slice = my_str.as_str();
let letter_h = my_str.truncate(1);
my_str.len();
my_str.pop();
```

- Conversion functions to get raw bytes and a string primitive
- Many core String functions are the same as the core Vec functions

```
let lo = my_str.split_off(3);
```

- Split a string at a given byte index

```
my_str.push_str(" World");
my_str.insert_str(0, "Well, ");
```

- Insert primitive strings into a String container

```
let foobar = "foobar";

let my_ptr = foobar.as_ptr();


let char_iter = foobar.char_indices();


foobar.find("foo"); // Returns Some(0)


let foo = foobar.get(0..4);

let mut foo2 = foobar.get_mut(0..4);

let trimmed = foobar.trim();


let three_times = foobar.repeat(3);
```

- Get a pointer to the first byte of the string slice
- Returns an iterator which will iterate over each char/index in the string slice
- Return the index of the first occurrence of the given string slice
- Get an optional substring from a string
- Remove leading/trailing whitespace
- Repeating a string slice – turning it into a String

Managing Collections In Memory

Heap-Allocated Data Structures/Types

Vec

HashMap

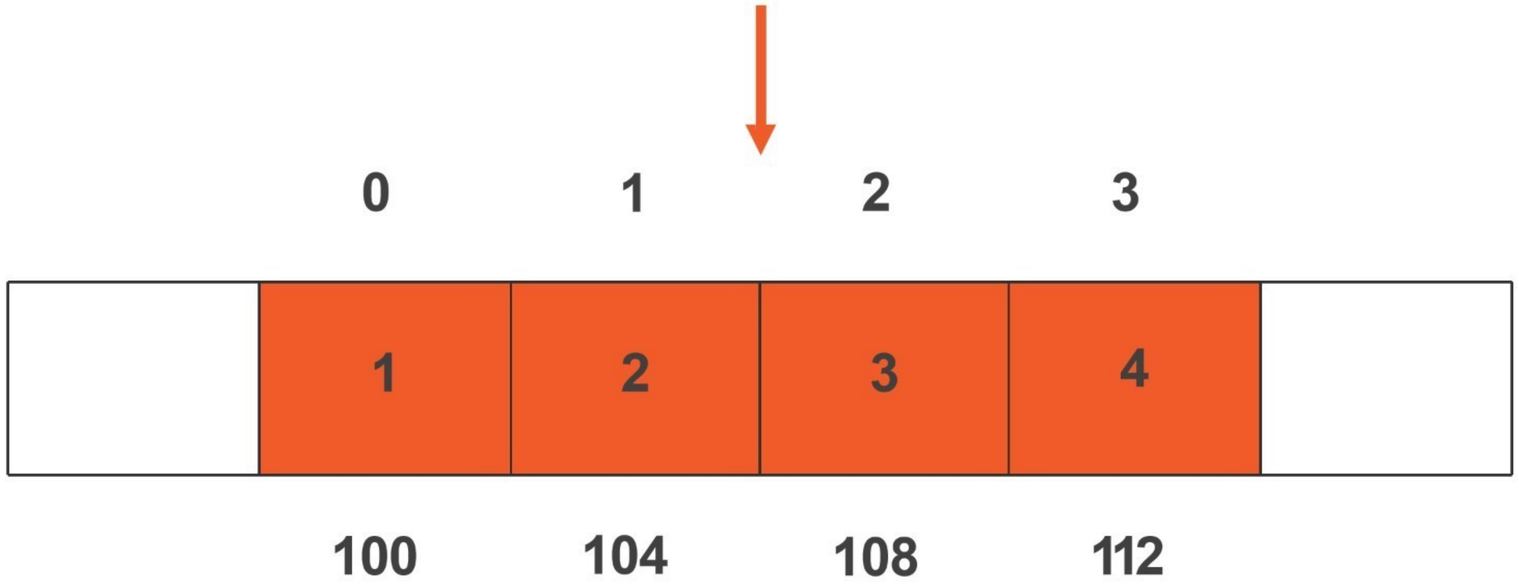
String

Contiguous Heap Allocation

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

Contiguous Heap Allocation

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



Arrays behind the scenes!

```
let mut my_vec = Vec::from([1, 2, 3]);

my_vec.capacity(); // three bytes

my_vec.push(4);

my_vec.capacity(); // four bytes or more
```

- Let's say, for example's sake, that the array backing this vector was allocated 3 bytes of memory on the heap

- We add a new value since vectors can grow in size dynamically!

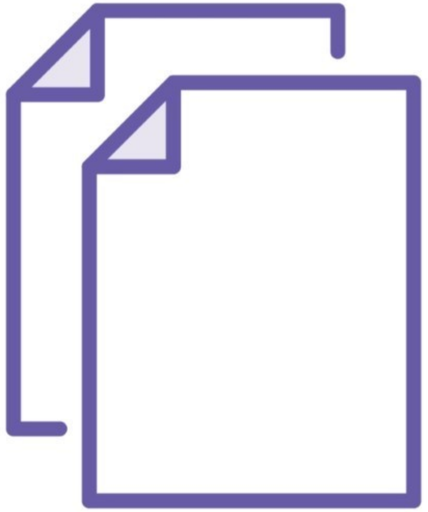
- But wait, arrays are fixed in size!
- A new array must be allocated memory behind the scenes – enough to fit the new data being added.

Moving, Copying and Cloning Collections

Moves and Copy/Clone Traits

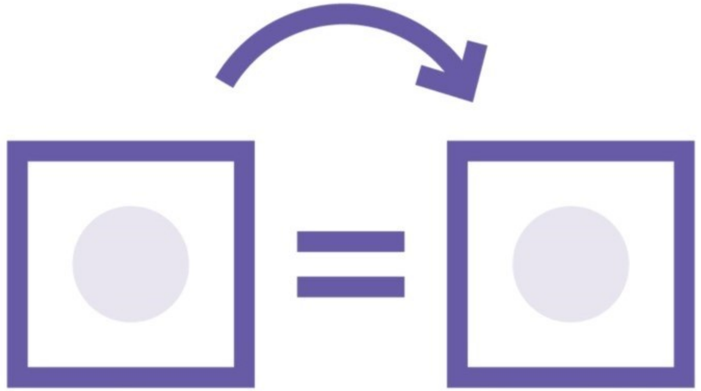
Move

Represents a simple bit-wise copy – a transfer of ownership.



Copy

A trait for types that do not own other resources.



Clone

A supertrait of copy – good for “deep” copying.

Collections can be moved and cloned but notcopied!

```
// Moving/Cloning Collections

let my_vec = vec![2, 4, 6];

let moved_vec = my_vec;

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

let my_clone = my_other_vec.clone();
```

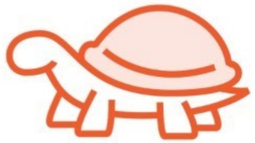
- Initializing a Vec using the ”vec!” macro
- Accessing “my_vec” now causes the code to not compile as ownership has moved to “moved_vec”
- Create a second vector
- Cloning has reproduced the buffer pointer, capacity and length information and has allocated memory for a new, deep copied vector on the heap

Applies to HashMap and String too!

Capacity Management

Capacity Management Fundamentals

Arrays are not always the exact “right” size at any point in time



Having an array always at the “right” size is inefficient!

Amortized allocation strategy

Allocation with extraspace!

Primary Capacity Functions

with_capacity() – just enough space for a given # of elements

reserve() – reserve extra space prior to new items being added

shrink_to_fit() – regain memory by shrinking the backing array

capacity() – gather capacity information from a collection

File I/O

Input Output with files

std::fs – the filesystem module

std::io – input/output module

Primary File Operations

std::fs::File::open

std::fs::File::create

std::fs::remove_file

File (Rust struct)

File struct

wrapper

The Rust "File" struct is a wrapper around an opened file. This struct wraps a file descriptor (file handle) and gives either read and/or write access to the opened file.

handle

access to

Thread

File :: open

Citation: Rust Documentation

need to deal with access parameters

OpenOptions – custom fileaccess permissions

File I/O

BufReader

BufWriter

The “write!” and “writeln!” macros

Reading Files

```
let mut foo_str = String::new();
let file_path = Path::new("foo.txt");

// Put the contents of the foo.txt file into a String
let mut foo_file = File::open(&file_path)?;
foo_file.read_to_string(&mut foo_str)?;
```

Reading Files

```
let file_path = Path::new("foo.txt");

// Read the contents of the foo.txt file line by line using BufReader
let mut foo_file = File::open(&file_path)?;

let buffer = BufReader::new(foo_file);

for line in buffer.lines() {
    println!("Current line: {}", line?);
}
```

When the File struct is dropped, the file descriptor/handle is cleaned up!

Writing Files

```
let file_path = Path::new("foo.txt");

let mut foo_file = File::create(file_path)?;

write!(foot_file, "foo text!")?;
```

Writing Files

```
let file_path = Path::new("foo.txt");

let mut foo_file = File::create(file_path)?;

let mut buffered_writer = BufWriter::new(foo_file);

writeln!(buffered_writer, "Write this text to the file!")?;

// Important to use flush explicitly with buffers so that errors get bubbled up
buffered_writer.flush()?;
```

Opening Files With the Right Access Permissions

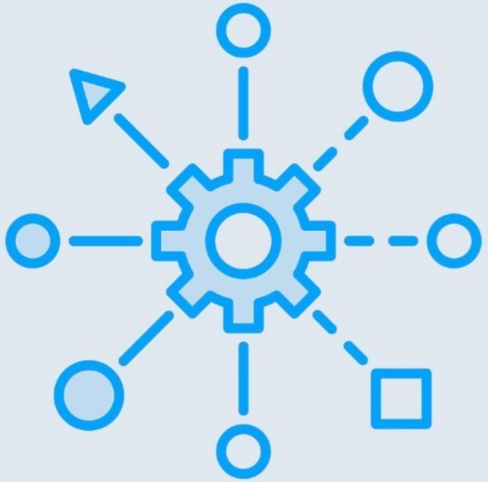
```
let mut my_file = OpenOptions::new()
                    .read(true)
                    .append(true) // also calls .write(true)
                    .create(true)
                    .open("/path/to/my_file.txt")?;
```

Concurrency and Rust

Concurrent & Parallel

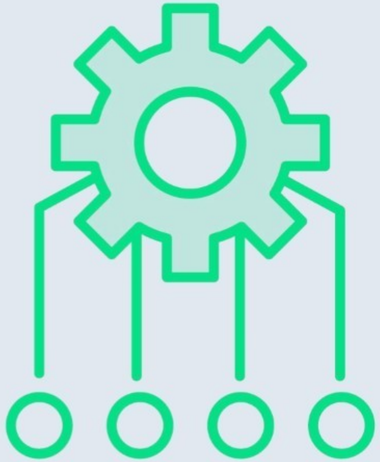
Different parts of a program running independently

Concurrency vs. Parallelism



Concurrency

A programming environment where multiple parts of a program run independent of one another



Parallelism

A programming environment where multiple parts of a program run at the same time.

Rust's Concurrency Goals

Unify safe memory handling with preventing concurrency bugs

Catch concurrency bugs at compile time – not runtime

Make writing concurrent code more simple

“Fearless Concurrency”

Using Threads

Thread

You can think of a thread as the smallest independently executable portion of a program.

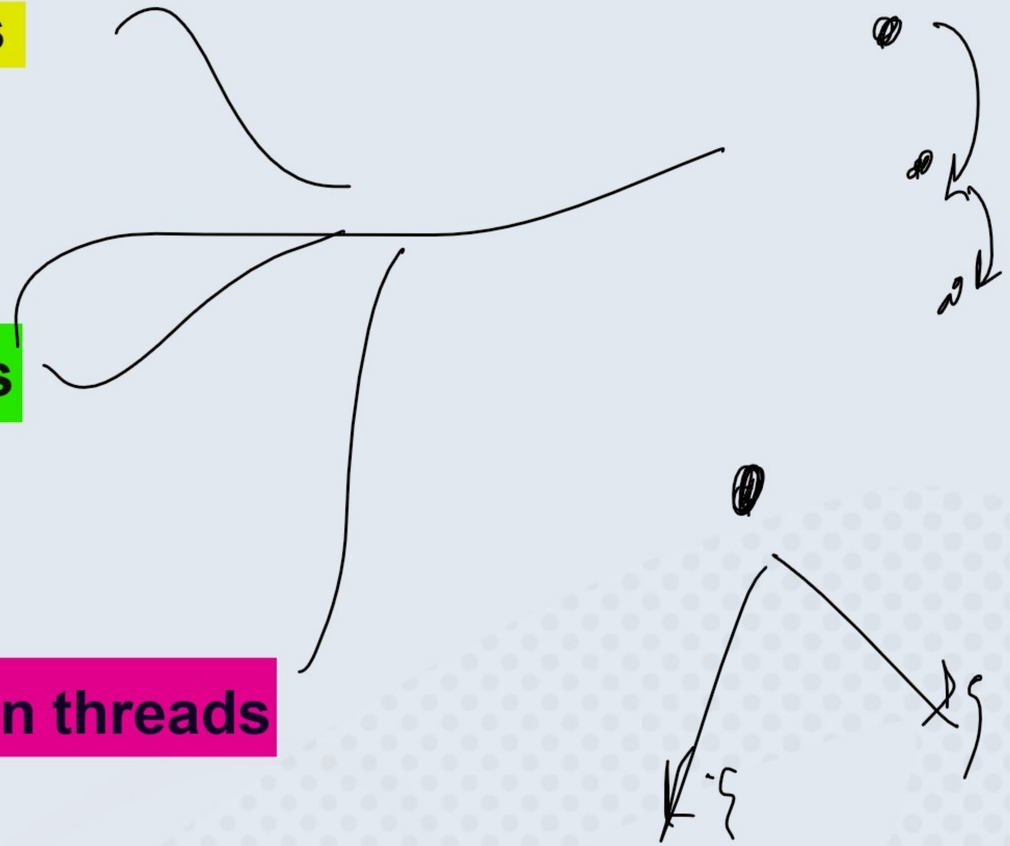
Thread Basics

How new threads are created

Spawning new threads

Joining thread handles

Moving values between threads



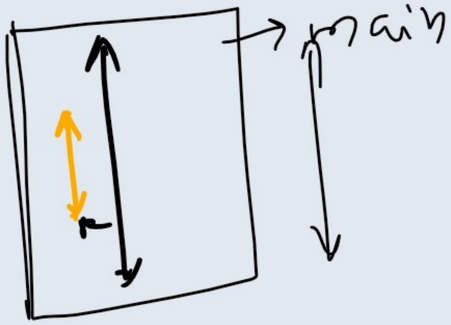
Spawning Threads

Spwan(|| { })

```
use std::thread;
```

```
thread::spawn(|| {
  println!("This is a log coming from within a separate thread!");
});

println!("This is a log coming from within the main thread!");
```

Waiting for Threads to Finish



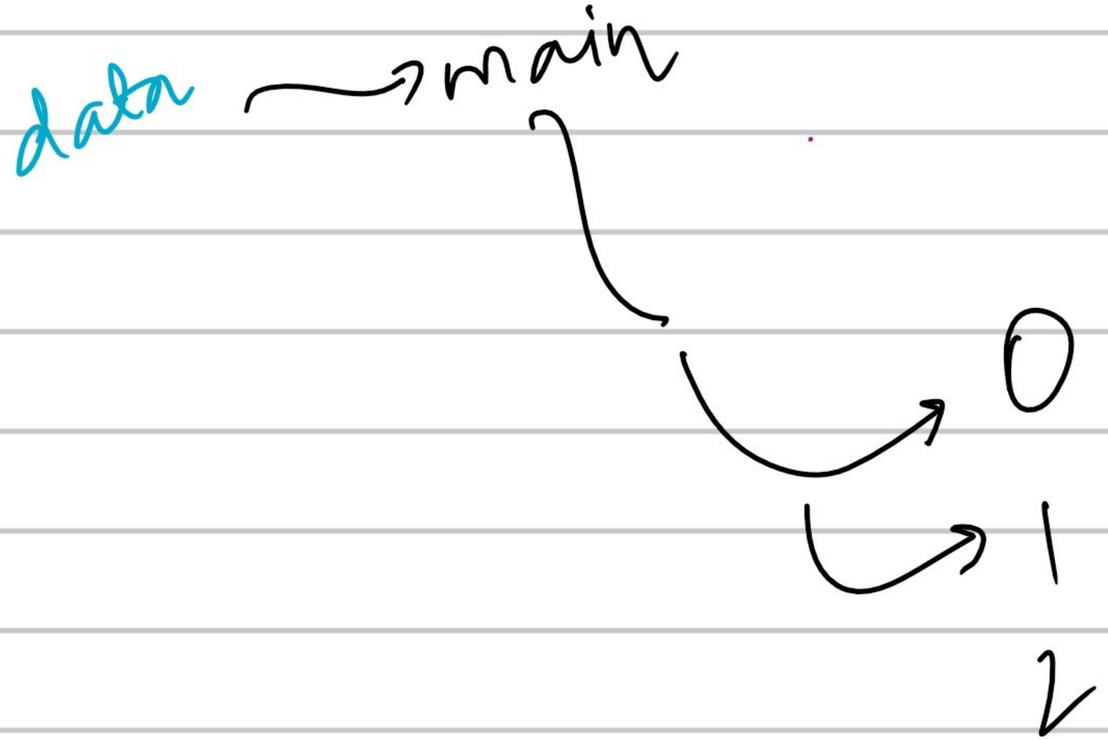

```
use std::thread;

let thread_handle = thread::spawn(||
  {  for num in 1..30 {
     println!("This is a log coming from within a separate thread!");
     thread::sleep(Duration::from_millis(1));
                                                                  }
                                                                 });
```

join

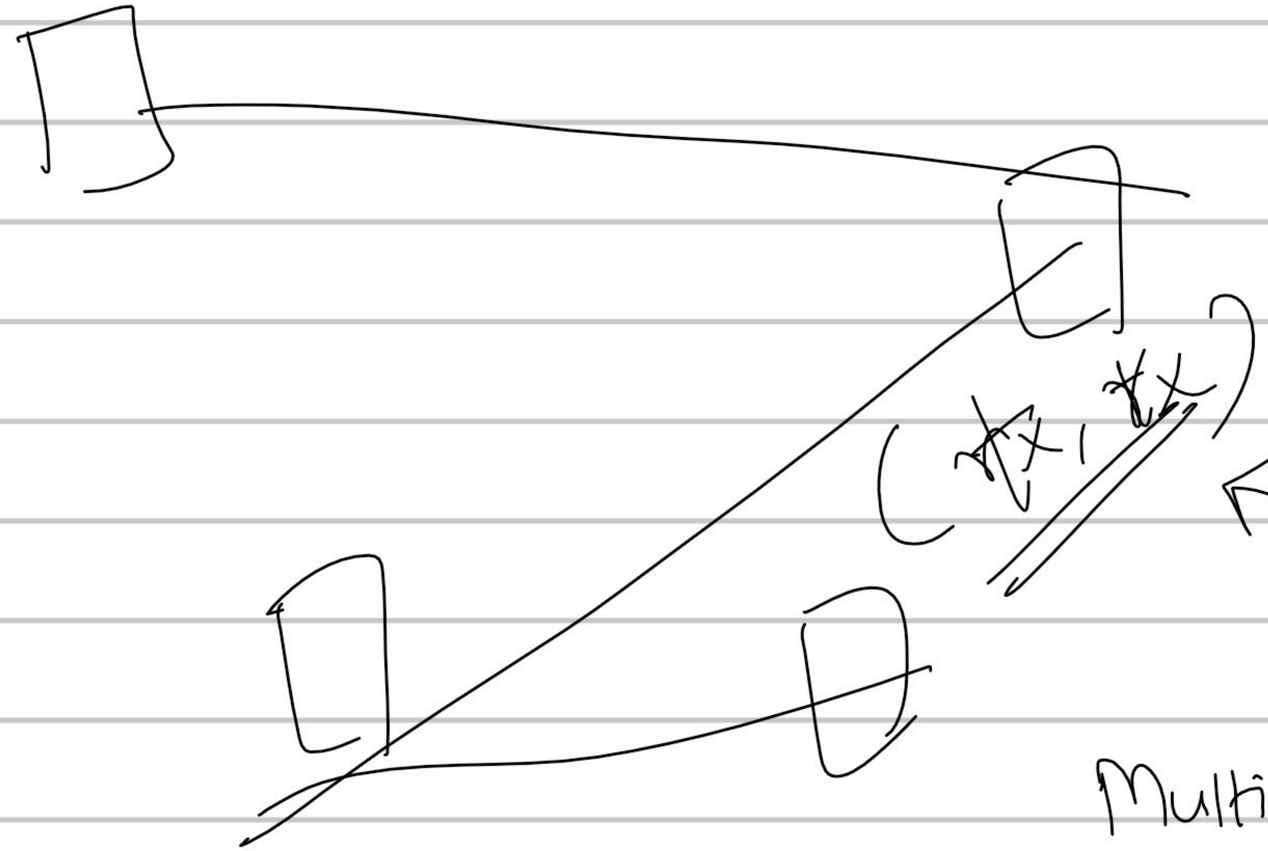
```
thread_handle.join().unwrap();
```

Transferring Values to Threads Using Move Closures

```
let my_str = String::from("Hello World!");

let thread_handle = thread::spawn(move || {
  println!("This thread now owns this string: {}", my_str);
});

thread_handle.join().unwrap();
```




```
let mut data = vec![ ]

for i in 0..3 {

    thread::spawn(move||
        {
        }
    );

}
```



Managing different threads

mpsc ::channel();

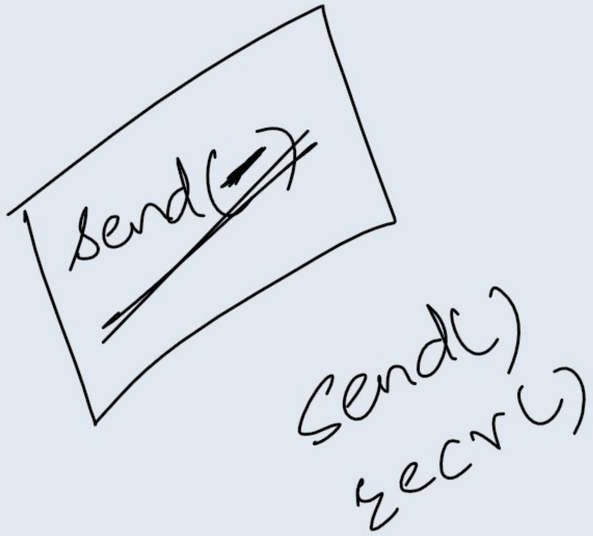
Multiple Producer
Single Consumer

Message Passing Using Channels

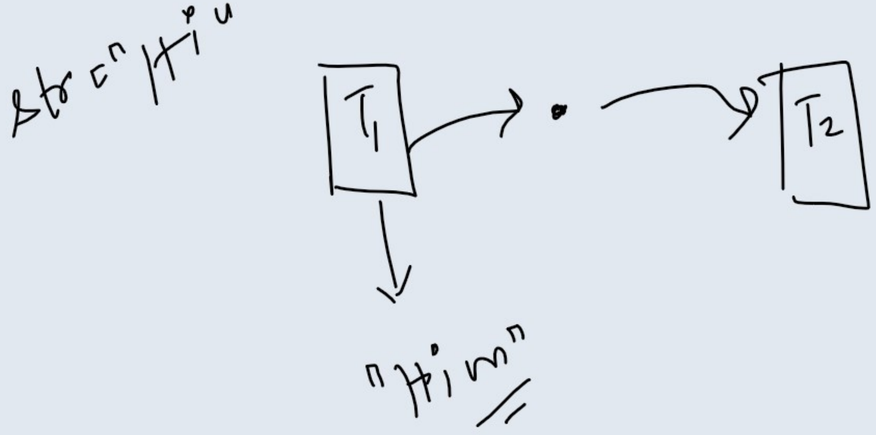
A way of sending data from one thread to another

Message Passing Between Threads Using Channels

```
use std::sync::mpsc;
use std::thread;

// MPSC: Multiple Producers, Single Consumer
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
  let my_str = String::from("Hello Other Thread!");
  tx.send(my_str).unwrap();
});

let received_msg = rx.recv().unwrap();
println!("Got this message from a different thread: {}", received_msg);
```



Safe Concurrency With Channels and Move Closures




```
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
  let my_str = String::from("Hello Other Thread!");
  tx.send(my_str).unwrap();

  // THIS LINE DOES NOT COMPILE! Ownership of "my_str" has moved
  println!("We sent this message: {}", my_str);
});

let received_msg = rx.recv().unwrap();
println!("Got this message from a different thread: {}", received_msg);
```

Shared-state Concurrency

Mutex

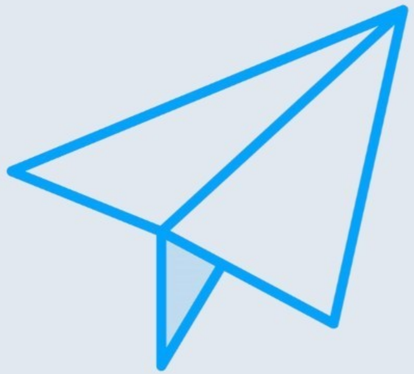
Short for “mutual exclusion”, mutexes make accessing shared data safe. Mutexes ensure that pieces of shared data are locked while being accessed by other threads. This makes your writes and reads consistent.

Shared-state Concurrency Using Mutexes

```
use std::sync::Mutex;

let my_mutex = Mutex::new(42);

{
  let mut custom_data_ref = my_mutex.lock().unwrap();
  *custom_data_ref = 12;
}

// The lock release happens automatically when the mutex goes out of scope!
```

Concurrency Traits

Concurrency Traits



Send

This trait allows transfer of ownership between threads

Sync

This trait marks a type has safely usable between multiple threads

You don’t have to manually implement these traits!

Unit Test

cargo test

vs

```
#[cfg(test)]
mod tests {

}
```

Structuring Rust Code

module
crate
package

A Rust module helps organize structs, enums, traits and functions into cohesive units

Using Modules and Paths

mod coffee{

```
// inline private module
mod coffee {

  // Coffee module code here…

}
```

}

```
// Public module

pub mod coffee;
```

coffee → mod.rs

src → coffee.rs

Using Modules and Paths

This declares a public module – if this module is in the root crate, the compiler will look for the code associated with this module in the "src/coffee.rs" file. If that file doesn't exist, the compiler will look at the "src/coffee/mod.rs" file.

Modules and “use”

```
pub mod coffee {

  pub struct Coffee {}
  pub struct CoffeeMachine {}
  pub struct CoffeeCup

}
```

Coffee::Coffee

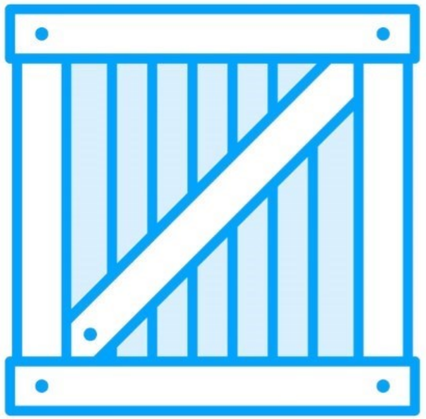
```
use crate::coffee::Coffee;

use create::coffee::{Coffee, CoffeeMachine};

use crate::coffee::*;
```

Using ”super” for Relative Paths

```
// Outer module code…

fn make_coffee() {}

// submodule
mod espresso {

  fn start_hybrid_mode()
    {
    super::make_coffee();
  }

}
```

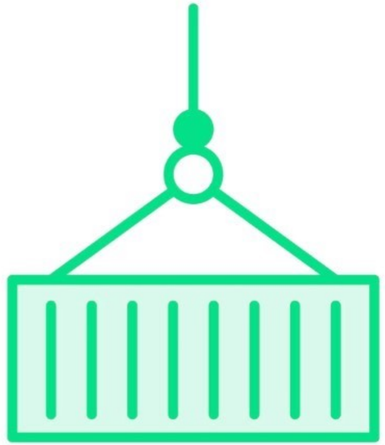
Crates and Packages

Organizing Rust Programs



Crates

A grouping of modules that produces either a library or an executable

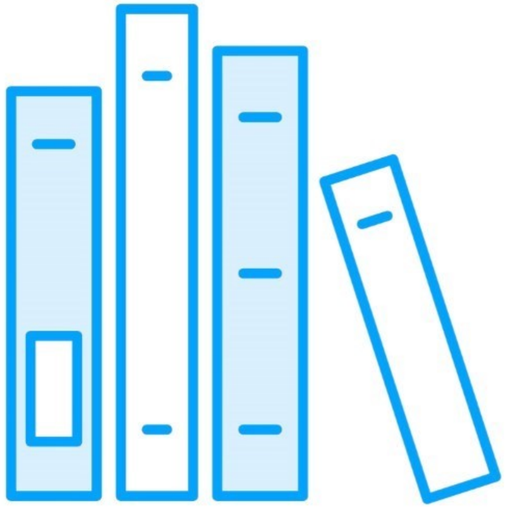


Packages

Used by Cargo, the Rust package manager, helps work with crates

Sometimes packages are referred to as crates!

Two Types of Crates



Library Crate

A crate that contains code meant to be consumed by other Rust projects

Binary Crate

A crate that is compiled to an executable

A package can contain ”x” number of binary crates but only 1library crate

Example Package Cargo.toml File

```
[package]
name = “test”
version = “0.1.0”
edition = “2021”
```

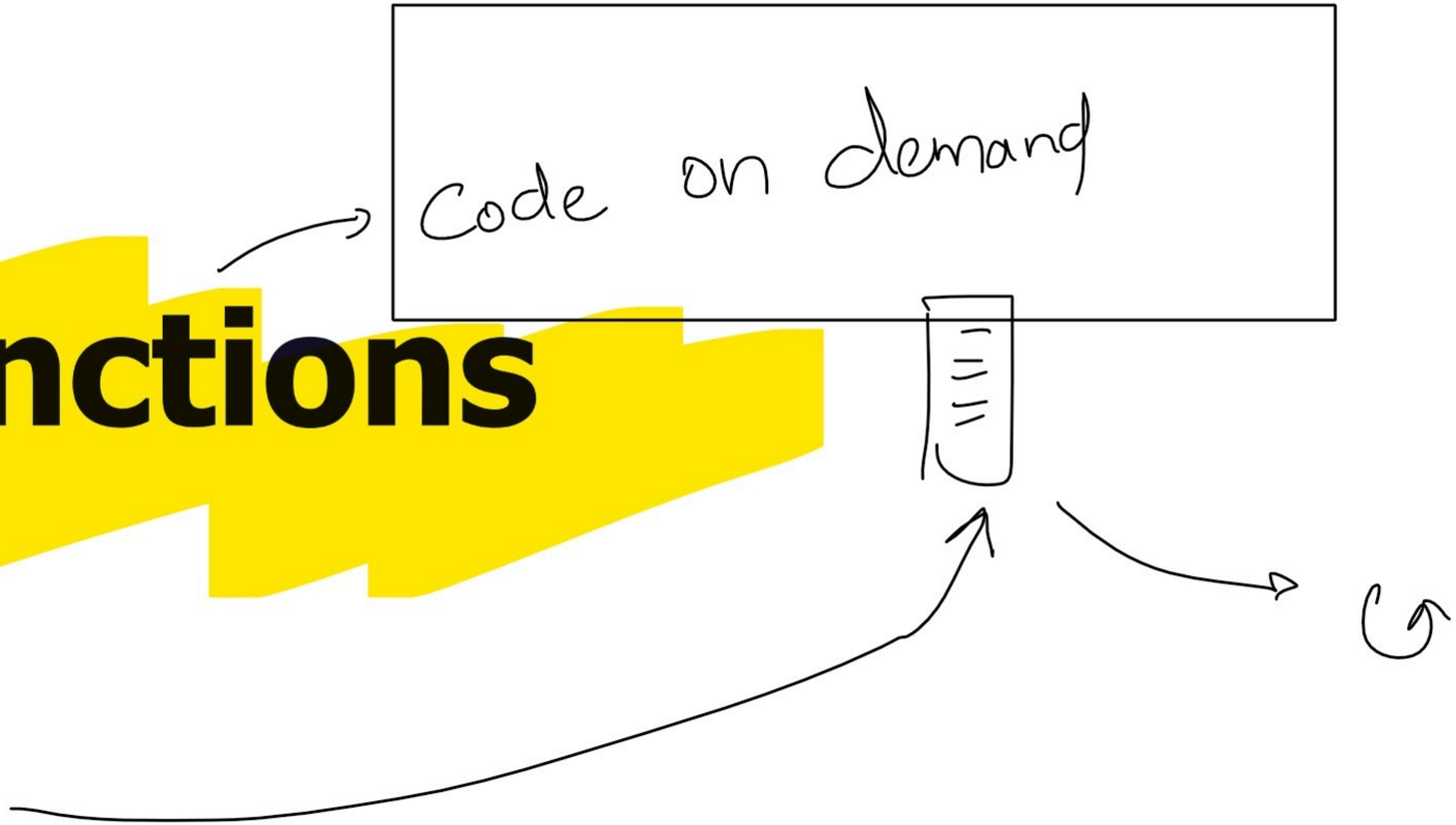
cargo build

cargo build --release

Third-party Dependencies

```
[package]
name = “test”
version = “0.1.0”
edition = “2021”

[dependencies]
time = “0.1.12”
regex = { git = https://github.com/rust-lang/regex.git }
```

Writing Functions



Functions in Rust

```
fn add (a:i32, b:i32) -> i32 {
    a+b
}
```

Name

Parameters/Arguments

Return type

Body

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
let result = add(1, 2);
```

No semi-colon & no ~~ret~~ return keyword

If you use return keyword ;

Function Anatomy

Functions are comprised of four primary parts: a name, parameters, a return type, and a body. The last expression evaluated within the function body becomes the return value.

A function is called by passing in a comma-delimited list of variable names enclosed by parentheses.

Function Anatomy “Gotcha”

```
fn add(a: i32, b: i32) -> i32 {
    a + b; // This will not compile because of the semicolon
}

------------

fn add(a: i32, b: i32) -> i32 {
    return a + b; // Adding a return statement or removing the semicolon works
}
```

Returning Early From Within a Function

```
fn add_or_subtract(a: i32, b: i32) -> i32
    {  if (a <= b) {
        return a + b;
    }

    a - b
}
```

Functions Without An Explicit Return Type

Unit → when returning nothing

↓

void

```
// This function returns nothing - it just performs a side effect
fn log(msg: &str) {
    println!("{}", msg);
}
```

Function names, like variable names, use “snake_case”

Using Functions

Rust Function Categories

Normal named functions

Named Functions

no name
no body

| assign to a variable
Pass it to a function

Anonymous Functions (Closures)

lambdas

my-arr.get()

Methods

functions assigned to a struct

String::from

Associated Functions

static functions

struct →

Named Functions

```
// Simple, named function

fn add(a: i32, b: i32) -> i32
    {  a + b
}
```

Anonymous Function (Closure)

Syntax

| parameter1, parameter2, . . . | expression

```
let my_anon_fn = |a: i32, b: i32| if a > b { a + b } else { a - b };

my_anon_fn(1, 2);
```

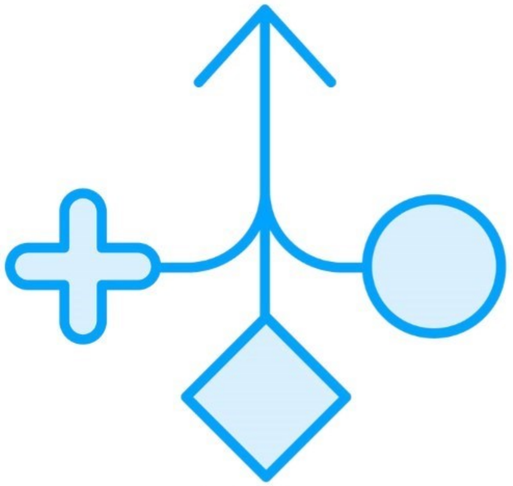
Anonymous functions can omit parameter types and returntypes.

```
let print_text = || println!("Defining Closure");
```

Closures and Scope

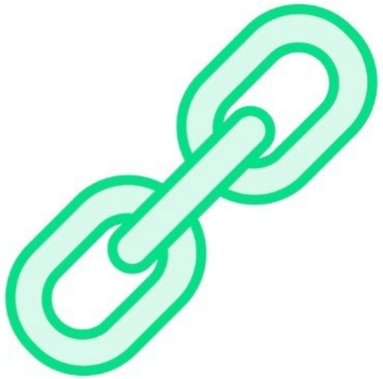
```
let my_num = 42;

// This function has access to a variable defined outside of its own scope
let my_anon_fn = |a: i32| if a < 0 { a + my_num } else { a - my_num };

my_anon_fn(1);
```

Function Associated With a Type



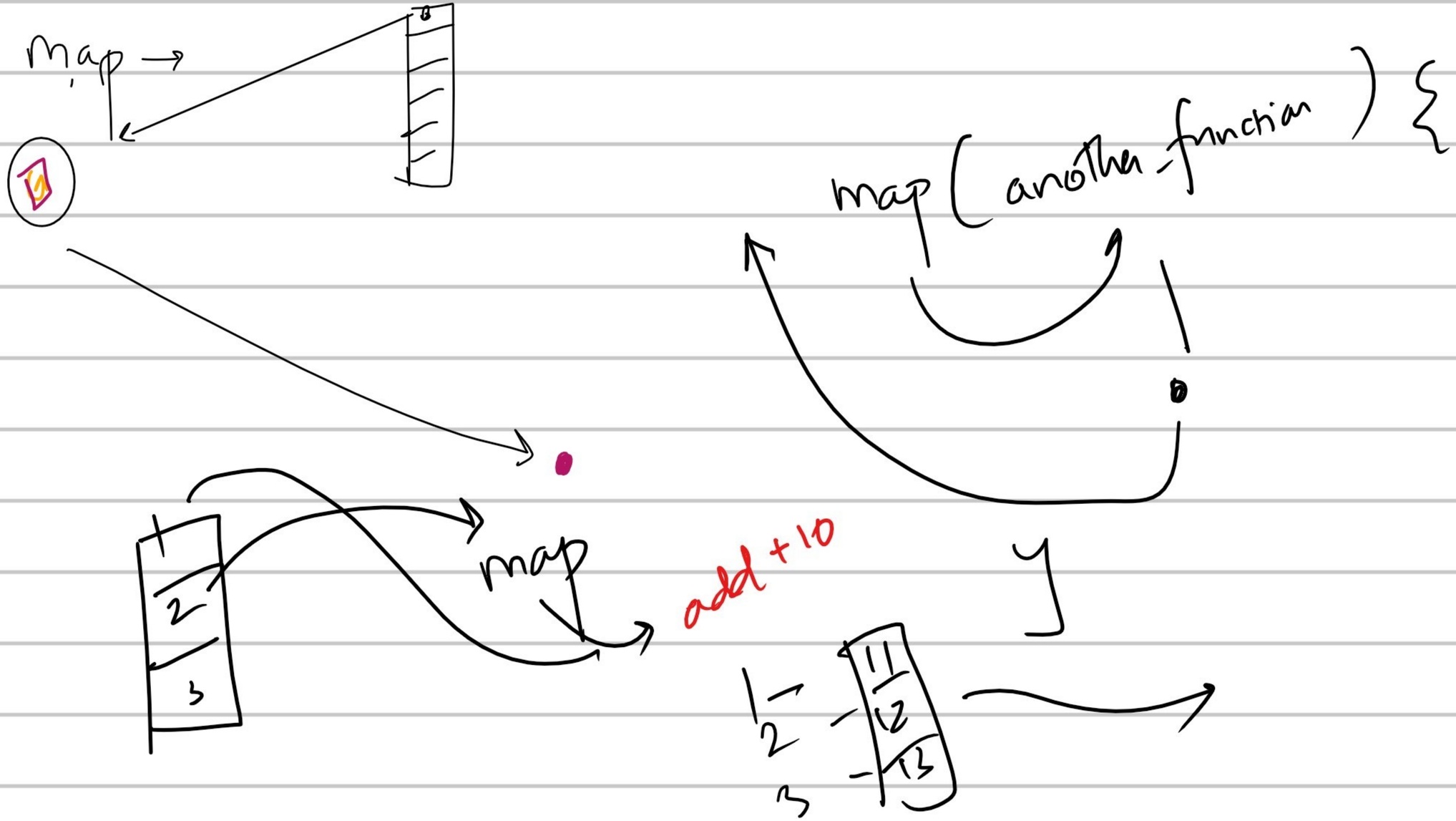
Methods

A method is a function that is associated with an instance of a type.



Associated Functions

An associated function is a function that is directly usable from a type itself


map →
map (another_function) {
map
add +10
1
2
3
11
12
13

Methods

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

impl Point {
    fn print(&self) {
        println!("X: {} | Y: {}", self.x, self.y);
    }

    fn add_one(&mut self)
        {  self.x += 1;
        self.y += 1;
    }
}
```

impl Point {

it will require an instance of the Point

&self

impl → used to implement functionality for a type

Methods always take “self” asthe first parameter

Associated Functions

```
struct Coffee
    {  temp:
    i32,  name:
    &str
}

impl Coffee {
   fn new(temp: i32, name: &str) -> Coffee
       {  Coffee {
            temp: temp,
            name: name
       }
   }
}
```

we do not need an instance to call associated functions

Associated functions do not require a reference to aparticular instance

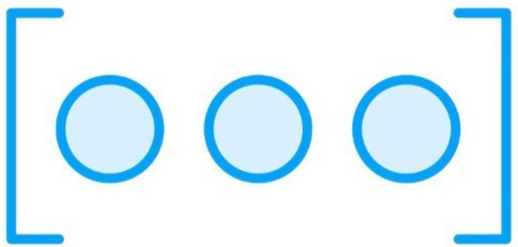
Major Collection Categories

Sequences

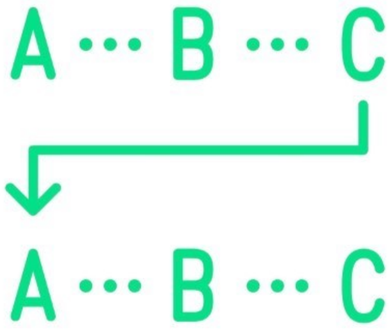
Maps

Sets

Sequence Types



Vec – a generic and growable sequence of data

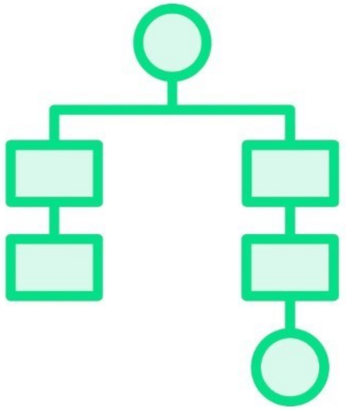


VecDeque – a double-ended queue

LinkedList – a doubly-linked list containing a series of nodes

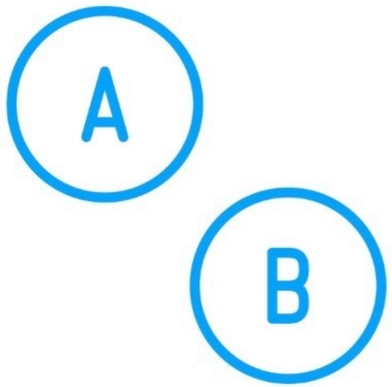
Map Types

HashMap
A key-value store



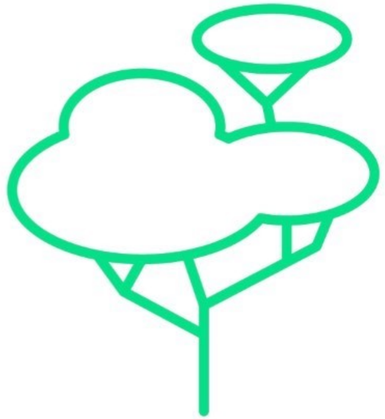
BTreeMap
A map optimized for search

Set Types



HashSet

A set containing no duplicates implemented as a HashMap



BTreeSet

An ordered set – optimized for search

Fundamental Collections

Vec

HashMap

Vec Key Features

Size can grow at runtime

Memory is allocated on the heap

Generic, homogenous data structure

Vec

```
// Creating vectors
let mut vec_a: Vec<i32> = Vec::new();

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

let vec_c = Vec::from([1, 2, 3]);
```

- Stores elements by key-value pairs
- Fast access of elements
- Great for in-memory caches
- Stores data on the heap

HashMap

```
// Creating hashmaps
let mut map_a: HashMap<&str, i32> = HashMap::new();

let map_b = HashMap::from([(“Hello”, 1)]);
```

Iterators

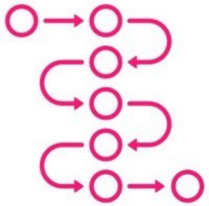
Iterator

An iterator is an abstraction around accessing elements in a collection.

Iterator Key Points

Iterators are lazy

Many collections like Vec use iterators



You can create your own iterators via the Iterator trait

Implicit Iterators

```
let my_nums = vec![1, 2, 3];
for num in my_nums {
    println!(" Number: {}", num);
}

// Behind the scenes, the code above is using iterators implicitly
// The "behind-the-scenes" code might look like the following
let my_nums = vec![1, 2, 3];

let nums_iterator = my_nums.iter();
for num in nums_iterator {
    println!(" Number: {}", num);
}
```

The Laziness of Iterators

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

// This doesn’t really do anything until you start consuming the iterator
let nums_iterator = my_nums.iter();
```

```
let my_nums = vec![1, 2, 3];
for num in my_nums {
    println!(" Number: {}", num);
}

let my_map = HashMap::from([("key1", 1)]);
for (key, value) in my_map {
    println!(" Number: {}", value);
}
```

Custom Iterators

You can create your own iterators by implementing the Iterator trait that Rust provides. These can be used to create custom iterators for your own collections.

For example, the "for/in" syntax can be used on both Vec and HashMap in the same way although the internals of how these iterations happen look different.

Error Handling and Debugging

Rust Error Concepts

Recoverable Errors

A Rust program should recover from these errors in some way and alert the user

Unrecoverable Errors

In case of these errors, the Rust program should immediately terminate

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

The Result Type

The Result type allows calling code to appropriately handle any potential recoverable errors that might occur.

Recovering from Errors with the Result Type

```
let open_file_result = File::open("test.txt");

match open_file_result {
    Ok(testFile) => println!("We successfully opened the file!"),
    Err(err) => println!("Can't open this file! Reason: {:?}", err)
}
```

Unrecoverable Errors and “panic!”

panic! → crash your code

SIGNAL BUGS

UNRECOVERABLE

```
let open_file_result = File::open(“test.txt”);

match open_file_result {
    Ok(testFile) => println!(“We successfully opened the file!”),
    Err(err) => panic!(“Can’t open this file! Reason: {:?}”, err)
}
```

Use panic! when you don’t want to give the calling code/user a chance to recover!

Propagating Errors with the Question Mark Operator

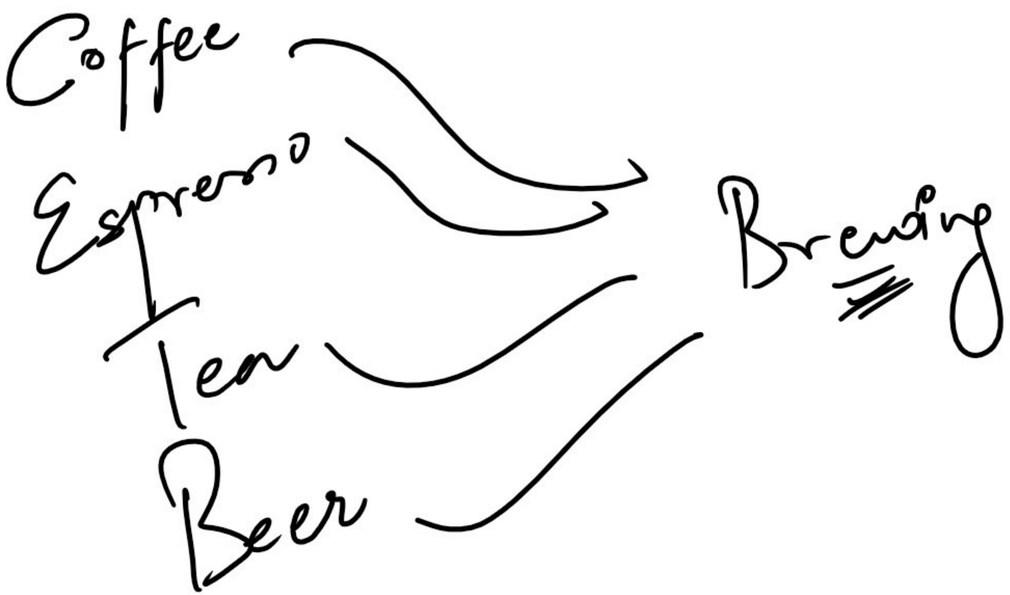
? → consise error handling
→ can be used inside a function returning Result or Option

```
fn multiply_str_by_ten(a: str&) -> Result<i32, ParseIntError> {
    let parsed_a: i32 = a.parse()?; // See how this unwraps the value?
    parsed_a * 10
}

multiply_str_by_ten("100"); // Returns a Result that is Ok(1000)
multiply_str_by_ten("abc"); // Returns an Error! The "?" operator propagates it
```

are like interface

Using Traits and Generics

Traits define shared behavior



Example Trait

Total

Student, Describe
String
Coffee

```
// Any type that implements this trait can calculate a total of some kind
pub trait Total {
    fn calc_total(&self) -> i64;
}

// Any type that implements this trait can calculate a description
pub trait Describe {
    fn gen_description(&self) -> String;
}
```

Using Traits

```
struct Snack
  {  name: String;
  item_number: i32;
  cost: f64;
}

struct Coffee
  {  name:
  String;
  cost: f64;
  size: String;
}

impl Describe for Snack {
  fn get_description(&self) -> String {
    format!("{} / Item #: {} / Cost: {}", self.name, self.item_number, self.cost)
  }
}

impl Describe for Coffee {
  fn get_description(&self) -> String {
    format!("{} {} costing {}", self.size, self.name, self.cost)
  }
}
```

Using Generics

Abstract types!

Why Generics?

Help reduce code duplication

More type safety - compile time guarantees

Flexible code but restrictive as necessary

Generic Structs

```
struct Coffee<'a, T, U>
  {  name: &'a str;
  cost: T;
  size: U;
}

let my_coffee = Coffee { name: "Drip", cost: 4.99, size: 3 };

let my_other_coffee = Coffee { name: "Drip", cost: 3.50, size: "Grande" };
```

Generic Enums

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

Generic Functions with Type Parameters

```
fn best<T>(items: Vec<T>) -> &T
  {  let mut best = &items[0];

  for item in items {
    if item.calc_rating() > best.calc_rating() {
      best = item;
    }
  }

  best
}
```

Enforncing Generic Type Behavior with Trait Bounds

```
pub trait Rate {
  fn calc_rating(&self) -> i32;
}

fn best<T: Rate>(items: Vec<T>) -> &T
  {  let mut best = &items[0];

  for item in items {
    if item.calc_rating() > best.calc_rating {
      best = item;
    }
  }

  best
}
```

Object-oriented Programming in Rust

Key Aspects of an Object-oriented Language

Supports objects

Supports encapsulation

Supports inheritance

Rust Object-oriented Language Features

VS

Objects - structs/enums

Encapsulation - pub/private

Polymorphism - via traits

Unsupported

Inheritance - although traits/supertraits are a way around this

Rust Version of Objects – Data with Methods

```
struct Coffee
  {  name:
  &str;  cost:
  f64;  size:
  &str;
}

impl Coffee {
  fn log(&self) -> () {
    println!(”{} {} costing {}”, self.size, self.name, self.cost);
  }
}

let my_coffee = Coffee { name: “Drip”, cost: 3.50, size: “Small” };

my_coffee.log();
```

Encapsulation

```
pub struct CoffeeInventory
  {  coffees: Vec<Coffee>;
  pub total: f64;
}

impl CoffeeInventory {
  pub fn add_coffee(&mut self, coffee: &Coffee
    {  self.coffees.push(coffee);
    self.update_total();
  }

  fn update_total(&mut self) {
    self.total = self.total + self.coffees[self.coffees.len() - 1].cost;
  }
}
```

Polymorphism

```
pub trait Mammal {
  fn describe_fur(&self);
}

impl Mammal for Monkey
  {  fn
  describe_fur(&self) {
    println!("A monkey's fur commonly contains black, brown or grey tones.");
  }
}

impl Mammal for Bear {
  fn describe_fur(&self) {
    println!("A bear's fur is commonly thick and shaggy.");
  }
}

fn mammal_fur_is(mammal: Mammal)
  {  mammal.describe_fur();
}
```

Pseudo-inheritance Via Supertraits

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

trait Mammal: Animal {
  fn describe_skin(&self) -> String;
}
```

You can use Rust in a functional way or in an object-oriented fashion – its up to you!

Symbol	What it represents?
*	Zero or more occurrences of the character that preceeds this asterisk
.	A wild card that represents any character
\s	Represents whitespace
[pqr]	A single character which can be either a ‘p’ , ’q’ or an ‘r’
[a-d]	A single character that falls in the range ‘a-d’
[^pq]	A single character that is neither ‘p’ nor ‘q’
^pattern	^ is an anchor tag that represents the beginning of line
pattern$	$i s an anchor tag that represents the end of line

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ fooaaaabar
- ☑ fooabar
- ☑ foobar
- ☑ fooaabar
- ☒ fooxxxbar
- ☒ fooxbar

☑ foo	aaaa	bar
☑ foo	a	bar
☑ foo		bar
☑ foo	aa	bar

foo a* bar

a* - Zero or more occurrences o 'a' (The character just preceeding the asterisk)

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ fooabar
- ☑ fooxbar
- ☒ baryfoo
- ☒ foobar
- ☒ fooxybar
- ☑ foocbar

☑ foo	a	bar
☑ foo	x	bar
☑ foo	c	bar

foo . bar

. - Single wildcard . Can represent only ONE character [any] in single position

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

0 - N → *

- ☑ foobar
- ☒ barfoo
- ☑ fooabcbar
- ☑ fooxcbar
- ☒ barcbyfoo
- ☑ foozbar
- ☒ barafoo
- ☒ barabfoo

☑ foo		bar
☑ foo	abc	bar
☑ foo	xc	bar
☑ foo	z	bar

foo	.*	bar

.* - Zero or more occurrences of wildcard, which means zero or more occurrences of any character

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ fooxxxbar
- ☑ foo bar
- ☒ fooxbar
- ☒ fooxxbar
- ☑ foo bar
- ☑ foo bar
- ☑ foobar
- ☒ fooyyybar

☑ foo	<3 spaces>	bar
☑ foo	<1 spaces>	bar
☑ foo	<6 spaces>	bar
☑ foo	<0 spaces>	bar

foo \s* bar

\s represents whitespace. \s* represents zero or more occurrence of whitespace

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

f [fcl]

c

l

- ☑ foo
- ☒ moo
- ☑ coo
- ☒ doo
- ☒ poo
- ☑ loo
- ☒ boo
- ☒ hoo

☑ f	oo
☑ c	oo
☑ l	oo

[fcl] oo

[abc] - Character class. One of the character inside the square brackets - a,b or c

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

^

- ☑ foo
- ☒ moo
- ☑ coo
- ☑ doo
- ☑ poo
- ☑ loo
- ☑ boo
- ☒ hoo

☒ m | oo

☒ h | oo

[^ m h] oo

[^abc] - Any character EXCEPT any of the one inside the square brackets, in a single position

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ joo
- ☒ moo
- ☑ koo
- ☑ loo
- ☒ poo
- ☒ zoo
- ☑ moo
- ☒ hoo

☑ j	oo
☑ k	oo
☑ l	oo
☑ m	oo

[jklm] oo

[j-m] oo

[a-c] - One of the character falling in the range given in square brackets

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ joo
☒ moo
☑ koo
☑ loo
☒ poo
☑ moo
☑ zoo
☒ hoo

☑ j oo
☑ k oo
☑ l oo
☑ m oo
☑ z oo

[jklmz] oo

[j-mz] oo

[a-cx] - One of the character falling in the range given in square brackets OR any of the other choices given in square brackets - a,b, c,x

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ joo
- ☒ moo
- ☑ Koo
- ☑ Loo
- ☒ poo
- ☑ moo
- ☑ zoo
- ☒ hoo

☑ j	oo
☑ K	oo
☑ L	oo
☑ m	oo
☑ z	oo

[j-m]-mz]oo

[a-cACx] - One of the character falling in the range given in square brackets OR any of the other choices given in square brackets - a,b, c,A,B,C,x

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ x#y

☑ x:y

☑ x.y

☒ x&y

☒ x%y

☑ x	#	y
☑ x	:	y
☑ x	.	y

x [#:.] y

If a '.' is inside square brackets, it need not to be escaped

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ x#y

☑ x:y

☑ x^y

☒ x&y

☒ x%y

☑ x	#	y
☑ x	:	y
☑ x	^	y

x [#:\^] y

If a '^ - ' is inside square brackets, it need to be escaped with backslash

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ x#y

☑ x\y

☑ x^y

☒ x&y

☒ x%y

☑ x	#	y
☑ x	\	y
☑ x	^	y

x [#\\^] y

If a '\ ' is inside square brackets, it need to be escaped with another backslash

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ foo bar baz
☒ bar foo baz
☒ baz foo bar
☒ bar baz foo
☑ foo baz bar
☒ baz bar foo

☑ foo	bar baz
☑ foo	baz bar

^foo.*

^ is a placeholder that signifies beginning of line. Inside [] a ^ means negation, but ouside brackets it acts as an anchor

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ foo bar baz
- ☒ bar foo baz
- ☑ baz foo bar
- ☒ bar baz foo
- ☑ foo baz bar
- ☒ baz bar foo

☑ baz	foo bar
☑ foo	baz bar

.* bar$

$ is a placeholder which signifies end of line

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ foo
- ☒ bar foo
- ☒ baz foo
- ☒ bar baz foo
- ☒ foo baz bar
- ☒ baz bar foo

☑ foo

^ foo $

^ signifies start of the line ; $ is a placeholder which signifies end of line

Symbol	What it represents?
+	One or more occurences of the character that preceeds + symbol
?	Zero or one occurence of the character that preceeds + symbol
pat1 | pat2	Match either the pattern pat1 or pattern pat2
()	Divides te pattern into groups
{m}	Exactly ‘m’ ocuurences of whatever preceedes
{m,n}	Atleast m and atmost n occurrences of whatever preceedes

1 - N

0 - 1

|

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ 834
- ☑ 519
- ☒ 4874
- ☒ 5
- ☒ 89
- ☒ 45687
- ☒ 25
- ☑ 645

☑ ^	[0-9][0-9][0-9]	$
☑ ^	[0-9][0-9][0-9]	$
☑ ^	[0-9][0-9][0-9]	$

^[0-9][0-9][0-9]$

^[0-9]{3}$

a{m} represents exactly 'm' repetations of whateer immediately precedess this

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ lion	☑ ^	[a-z]{4}	$
☑ tiger			
☒ leopard	☑ ^	[a-z]{5}	$
☒ fox			
☒ kangaroo	☑ ^	[a-z]{5}	$
☒ cat			
☑ mouse	☑ ^	[a-z]{6}	$
☑ cuckoo			
☑ deer	☑ ^	[a-z]{4}	$

^[a-z]{4,6}$

a{m,n} represents atleast 'm' and atmost 'n' repetations of whateer immediately precedess this

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ ha
- ☑ hahahahaha
- ☒ hahaha
- ☑ hahahaha
- ☒ haha
- ☑ hahahahahaha
- ☑ hahahahahahahaha
- ☑ hahahahahahahahaha

- ☑ ha{5}
- ☑ ha{4}
- ☑ ha{6}
- ☑ ha{8}
- ☑ ha{9}

(ha){4,9}

paranthesis is used for grouping and treating as a single entity
{m,} represents atleast m repeatations of whatever preceeds

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ fooaaaabar
☑ fooabar
☒ foobar
☒ fooxxxbar
☒ fooxbar

✓ foo aabar

☑ foo	aaaa	bar
☑ foo	a	bar
☑ foo	aa	bar

foo a+ bar

a+ One or more occurences of 'a'

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ https://website
☑ http://website
☒ httpss://website
☒ httpx://website
☒ httpxx://website

☑ http	s	://website
☑ http		://website

http s? ://website

a? Zero or One occurence of 'a'

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ redwood
- ☑ logwood
- ☒ sapwood
- ☒ rosewood
- ☑ plywood
- ☒ teakwood

☑ log	wood	
☑ ply	wood	

(ply | log) wood

(a|b) represents either a or b, where a and b can be multi character strings

`\A` - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
`\Athe`	`the sun`	Match
	`In the sun`	No match

`\Z` - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
`Python\Z`	`I like Python`	1 match
	`I like Python Programming`	No match
	`Python is fun.`	No match

`\b` - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
`\bfoo`	`football`	Match
	`a football`	Match
	`afootball`	No match
`foo\b`	`the foo`	Match
	`the afoo test`	Match
	`the afootest`	No match

`\B` - Opposite of `\b`. Matches if the specified characters are **not** at the beginning or end of a word.

Expression	String	Matched?
`\Bfoo`	`football`	No match
	`a football`	No match
	`afootball`	Match
`foo\B`	`the foo`	No match
	`the afoo test`	No match
	`the afootest`	Match

`\d` - Matches any decimal digit. Equivalent to `[0-9]`

Expression	String	Matched?
`\d`	`12abc3`	3 matches (at `12abc3`)
	`Python`	No match

`\D` - Matches any non-decimal digit. Equivalent to `[^0-9]`

Expression	String	Matched?
`\D`	`1ab34"50`	3 matches (at `1ab34"50`)
	`1345`	No match

`\s` - Matches where a string contains any whitespace character. Equivalent to `[ \t\n\r\f\v]`.

Expression	String	Matched?
`\s`	`Python RegEx`	1 match
	`PythonRegEx`	No match

`\S` - Matches where a string contains any non-whitespace character. Equivalent to `[^ \t\n\r\f\v]`.

Expression	String	Matched?
`\S`	`a b`	2 matches (at `a b`)
		No match

`\w` - Matches any alphanumeric character (digits and alphabets). Equivalent to `[a-zA-Z0-9_]`. By the way, underscore `_` is also considered an alphanumeric character.

Expression	String	Matched?
`\w`	`12&": ;c`	3 matches (at `12&": ;c`)
	`%"> !`	No match

`\W` - Matches any non-alphanumeric character. Equivalent to `[^a-zA-Z0-9_]`

Expression	String	Matched?
`\W`	`1a2%c`	1 match (at `1a2%c`)
	`Python`	No match

Email Pattern

1. It should have one @ symbol in between
2. Before @ it can be any number of a-z,A-Z, _,.
3. After @ there should be minimum one number of a-z,A-Z,0-9,-
4. After @ and previous combination there should be minimum one period(.) sign
5. It should end with either com,org or net

Indian Mobile Number Pattern

1. It should have 10 numbers
2. Starting number should be either 7,8, or 9 , 6