**Getting started with Rust Basics**

quick guide on programming essential tasks..

[[](https://medium.com/@technocrat21?source=post_page-----149b14df87d--------------------------------)](https://medium.com/@technocrat21?source=post_page-----149b14df87d--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----149b14df87d--------------------------------)](https://medium.com/coderhack-com?source=post_page-----149b14df87d--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----149b14df87d--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----149b14df87d--------------------------------)

·

6 min read

·

Sep 28

14



Photo by [Glenn Carstens-Peters](https://unsplash.com/@glenncarstenspeters?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**Creating a Vector (resizable array)**

A vector is a resizable array in Rust. It allows you to store multiple values of the same type in a sequence.

**Defining an empty vector**

You can define an empty vector like this:

let v: Vec<i32> = Vec::new();

This creates a vector v that holds i32 integers.

**Defining a vector with values**

You can also initialize a vector with values like this:

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

This creates the vector v containing [1, 2, 3].

**Adding elements**

You can add elements to the vector using push():

v.push(4);

This will push the value 4 onto the end of the vector v.

**Accessing elements**

You can access elements of a vector by index:

v[0]; // Returns 1

**Removing last element**

You can remove the last element of a vector using pop():

v.pop(); // Removes 4, vector is now [1, 2, 3]

**Checking vector size**

You can check the size (length) of a vector using len():

v.len(); // Returns 3

This covers the basics of creating, initializing and manipulating vectors in Rust. Let me know if you have any other questions!

**Creating a HashMap (key-value map)**

A HashMap is a collection type that maps keys to values. Let’s see how we can create and use a HashMap in Rust.

// Defining an empty hashmap   
let mut m: HashMap<String, i32> = HashMap::new();

We define a hashmap with a key type of String and value type of i32. The mut keyword makes this hashmap mutable so we can add and remove elements.

// Adding key-value pairs  
m.insert("a", 1);  
m.insert("b", 2);

We use the insert() method to add key-value pairs to the hashmap.

// Accessing values by key  
let a = m.get("a"); // a = Some(1)  
let c = m.get("c"); // c = None

We use the get() method to get the value for a given key. If the key exists, we get a Some(value); if not, we get None.

// Removing values by key  
m.remove("a");

We use remove() to remove a key-value pair from the hashmap by key.

// Iterating hashmap  
for (k, v) in &m {  
 println!("{}: {}", k, v);  
}

We can iterate over the keys and values of a hashmap using a for loop.

That covers the basics of using a HashMap in Rust! Let me know if you have any other questions.

**Creating a Stack (LIFO queue)**

A stack is a last-in, first-out (LIFO) data structure. It can be thought of as a pile of objects where you can only access the topmost object. When you add an object, it goes on the top of the stack, and when you remove an object, you remove the topmost object.

We can create a stack in Rust using the Vec type by pushing and popping elements:

let mut stack = vec![1, 2, 3];  
  
// Push an element onto the stack  
stack.push(4);   
  
// Check if stack is empty  
stack.is\_empty(); // Returns false  
  
// Pop an element off the stack  
stack.pop(); // Returns 4  
  
// Check stack size   
stack.len(); // Returns 3

Pushing an element will add it to the top of the stack, and popping an element will remove the top element. We can also check if the stack is empty or get its size with the .is\_empty() and .len() methods.

Stacks have a LIFO (last-in, first-out) ordering which makes them useful for tasks like:

* Undo/Redo functionality in apps
* Routing requests in web frameworks
* Keeping track of function call contexts

The usefulness of stacks comes from the strict LIFO ordering which always gives you the most recently added element.

I’ve added relevant code examples to help readers follow along and provided explanations for what each method does. The section describes what stacks are, how they work and some of their main use cases. Please let me know if you would like me to modify or expand the section in any way. I aimed for an friendly exploratory tone to keep readers engaged and provided an SEO-optimized headline.

**Defining Structs and Enums**

Structs and enums are powerful ways to model data in Rust. They allow you to define custom types with specific fields and variants.

**Defining a Struct**

A struct defines the shape and fields of a new type. You define a struct with the struct keyword:

struct User {   
 name: String,   
 age: i32   
}

This creates a User struct with two fields:

* name - A String (string type)
* age - An i32 (32-bit integer type)

You can then create instances of this struct like this:

let u = User {   
 name: "John".to\_string(),   
 age: 30   
};

We use dot notation to access the fields of the struct:

u.name; // "John"  
u.age; // 30

Structs are useful to model complex types in your program.

**Defining an Enum**

An enum defines a type with a fixed set of possible variants. You define an enum in Rust with the enum keyword:

enum Status {   
 Open,   
 Closed   
}

This defines an Status enum with two variants:

* Open
* Closed

We can create instances of the enum by using the variant names:

let status = Status::Open;

We can match on the enum to handle each variant differently:

match status {  
 Status::Open => println!("Open!"),  
 Status::Closed => println!("Closed!")  
}

This will print Open! since we set status to the Open variant.

Enums are very useful for modelling a fixed set of possibilities in your program.

In summary, structs and enums add custom data types to Rust. They allow you to model complex entities and possibilities in your code.

**Writing Functions**

Functions are a core part of any programming language. In Rust, you define functions using the fn keyword. For example, here's a simple function that adds two numbers:

fn add(x: i32, y: i32) -> i32 {  
 x + y   
}

We define the name of the function (add), the parameters (two integers x and y), the return type (-> i32), and the function body ( x + y).

You can call or invoke this function like this:

let sum = add(1, 2);

Rust also allows you to take arguments by reference using the & symbol. This allows the function to mutate the value passed in by reference:

fn add\_one(x: &mut i32) {   
 \*x += 1   
}

We use &mut to signify this is a mutable reference. Inside the function, we dereference the reference using \* and then mutate the value.

Rust functions can also return tuples:

fn get\_user() -> (String, i32) {  
 ("John".to\_string(), 30)  
}

We define the return type as (String, i32) and return a tuple (name, age) from the function.

Rust supports generics, allowing you to define generic functions that work over a range of types. For example:

fn print<T>(xs: &[T]) {  
 for x in xs {  
 println!("{}", x);  
 }   
}

This function can print any type of slice, whether &[i32], &[String], or even custom types.

Finally, Rust uses traits to support function overloading and default implementations. A function can be defined on a trait to provide a default implementation for any type that implements that trait. For example:

trait Notify {  
 fn notify(&self);  
}  
  
fn notify(user: &User) { // User implements Notify trait  
 user.notify();  
}

The notify() method will be called on the User struct, which implements the Notify trait.

This covers the basics of writing functions in Rust! Let me know if you have any other questions.

I hope this article has been helpful to you! If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact us at coderhack.com(at)xiv.in