

Міністерство освіти і науки України  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВО-МОГИЛЯНСЬКА АКАДЕМІЯ»  
Кафедра інформатики факультету інформатики

**Development of a course on learning the Rust programming  
language and its usage in developing DApps using Substrate framework**

**Текстова частина до курсової роботи  
за спеціальністю «Комп'ютерні науки» - 122**

**Керівник курсової роботи**

старший викладач

Гороховський К.С.

\_\_\_\_\_ (Підпис)

“ \_\_ ” \_\_\_\_\_ 2022 року

**Виконав студент**

КН-3 Михайленко О. І.

“ \_\_ ” \_\_\_\_\_ 2022 року

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

старший викладач

\_\_\_\_\_ Гороховський К.С.

„\_\_\_\_” \_\_\_\_\_ 2021 р.

## ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Михайленку Олександрю Ігоровичу

факультету інформатики 3 курсу бакалаврської програми

**ТЕМА: Development of a course on learning the Rust programming language and its usage in developing DApps using Substrate framework**

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Вступ

Розділ 1. Розробка курсу

Висновки

Список літератури

Додатки (за необхідністю)

Дата видачі „\_\_\_\_” \_\_\_\_\_ 2021 р.

Керівник \_\_\_\_\_

(підпис)

Завдання отримав \_\_\_\_\_

(підпис)

### Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	01.10.2021	
2.	Огляд літератури за темою роботи	01.11.2021	
3.	Проведення дослідження	01.12.2021	
4.	Написання програмного застосунку	01.01.2022	
5.	Написання текстової частини	04.04.2022	
6.	Захист курсової роботи	20.05.2022	

Студент \_\_\_\_\_

Керівник \_\_\_\_\_ “ \_\_\_\_ ” \_\_\_\_\_ 2022

## **Зміст**

<b>Календарний план виконання роботи</b>	<b>3</b>
<b>Chapter 1</b>	<b>6</b>
<b>Introduction</b>	<b>6</b>
<b>Welcome to the course!</b>	<b>6</b>
<b>This is going to be hard</b>	<b>7</b>
<b>This will not be graded</b>	<b>7</b>
<b>What will be covered</b>	<b>7</b>
<b>The motivation</b>	<b>8</b>
<b>Things you will acquire after the successful completion</b>	<b>8</b>
<b>The course is hard not just to be hard. It is meant to be rewarding.</b>	<b>8</b>
<b>Approximate learning plan</b>	<b>8</b>
<b>Week 1</b>	<b>8</b>
<b>Lecture: Meeting Rust</b>	<b>9</b>
<b>Practice: Rustlings</b>	<b>18</b>
<b>Week 2</b>	<b>19</b>
<b>Lecture: Advanced Rust</b>	<b>20</b>
<b>Lecture 2</b>	<b>29</b>
<b>Week 3</b>	<b>31</b>
<b>Lecture: Ethereum, smart contracts</b>	<b>31</b>
<b>Lecture: Substrate</b>	<b>39</b>
<b>Week 4</b>	<b>44</b>
<b>Practice: Substrate FRAME</b>	<b>44</b>

<b>Practice: runtime upgrades &amp; governance</b>	<b>45</b>
<b>Week 5</b>	<b>46</b>
<b>Lecture - ink! and smart contracts</b>	<b>46</b>
<b>QnA - Integrating a real-world application with blockchain &amp; QnA with Gautam Dhameja</b>	<b>46</b>
<b>Our logical conclusion</b>	<b>47</b>
<b>Literature used</b>	<b>47</b>

## Abstract

Substrate is a versatile blockchain development framework, based on the Rust programming language. While being well documented, Substrate is hard to learn without a thought-through approach – just taking on the documentation is not enough & will get the learner not much knowledge. Therefore, here we develop a course that explains the Rust programming language, basic blockchain development primitives, describes functioning models of different popular blockchains, and finally gives an overview of Substrate with some examples.

## Chapter 1

### Introduction

#### Welcome to the course!

Today marks the day we start learning Substrate and revising Rust. Our Rust revision will be rather long because we want to make sure everyone gets a firm grasp of the required concepts. For some, it will not be a revision. Instead, you will have to learn Rust from scratch. In two weeks. In three days. So prepare yourself for the ultimate glory! You'll have fun, hehe.

#### This is going to be hard

This course will not be easy. Be ready for some good commitment, as Substrate itself is hard and sometimes mundane (high build times). Nevertheless, we will try to make it as interesting as possible! The estimated commitment for this course is 5hrs a week. This

includes 2 hours for the two meetings a week (1 hour for a meeting) and 3 hours for the home tasks.

### **This will not be graded**

Although this course is very serious in making someone understand Substrate, it is not a certification, nor have I the right to certify people. The assignments will not be graded, and I recommend everyone find a pair and do peer reviews of the home tasks. I encourage discussing implementation details, general approach to the home tasks, and code details.

Since this course is not graded, you don't get to profit from cheating. As opposed to what they say at the universities (you're only harming yourself by cheating), this really is the case here. So, you can cheat, of course, I don't really care, but I guarantee you'll instantly lose motivation in this course.

### **What will be covered**

In this course, a couple of important things will be covered. This includes:

- Revision of Rust concepts, which are critical to understanding Substrate:
  - Rust's syntax
  - Ownership & Borrowing
  - Generics + Traits
  - Smart Pointers
  - Rust's memory model
  - Macros
- General blockchain development concepts:
  - Blocks
  - Crypto
  - PKI
  - Fees
  - Consensus algorithms
  - Intro to Smart Contracts
- Ethereum (revision):
  - General architecture
  - Smart contract execution platform architecture
  - Gas model
  - Use cases
- Substrate development key concepts:
  - FRAME
  - Runtime
  - Storage
  - Client libraries
  - Integration

- High-level substrate concepts:
  - Paraobjects
  - Accounts
  - PKI
  - Randomness
  - NPOS
  - Crypto
- Substrate-based smart contracts.

But definitely not limited to.

## The motivation

Why did we choose Substrate in the first place? Substrate is a great and very versatile blockchain development framework. I personally enjoy working with it, and this was one of the main motivations for teaching it :P The motivation to learn it are 1. High market demand 2. Reflection of all common blockchain development principles in one place (meaning endless space to practice those principles) 3. Despite a steep learning curve, unmatched ease of practical (as in business) uses 4. Modernity and increasing adoption

Just take a look at [teams who build in Substrate!](#)

## Things you will acquire after the successful completion

**The course is hard not just to be hard. It is meant to be rewarding.**

After completing the course successfully, you will learn a set of cutting-edge technologies (Substrate, Polkadot, ink!, Rust), one super important skill (blockchain development), one beautiful programming language (Rust), and a wider outlook on the current state of tech, Web3 understanding and most definitely a job among the most interesting Web3 companies around the world.

Next up: Parity company pitch, 4ire company pitch.

## Approximate learning plan

We plan this course to last 5 weeks.

So, the plan approximately goes like this:

### Week 1

- Introduction <- you're here
- Rust revision (basics)

Materials: [The Book](#), [Parity](#), [4ire](#), **Homework**: Complete chapters 1-10 of The Rust Programming Language

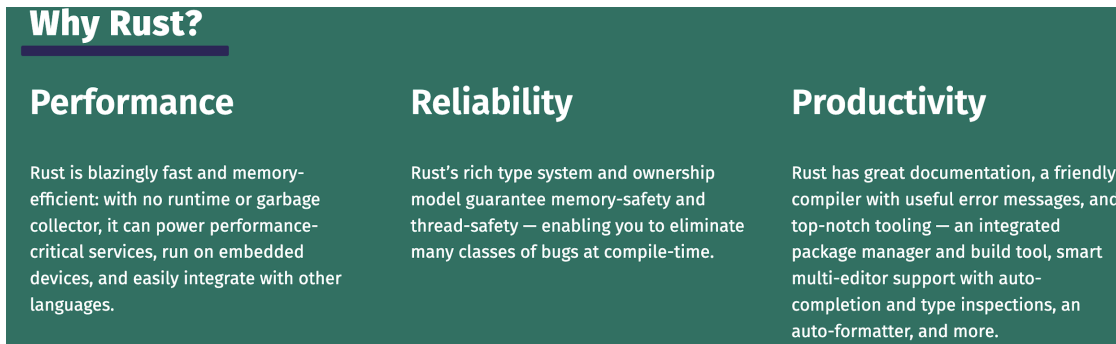


## *Lecture: Meeting Rust*

Alright! We're here. Today marks the day we start learning Rust & blockchain development. This course is very oriented toward self-education. Each week, we will have 2 meetings: a lecture and a practice. Moreover, you will get one homework for the week. I don't like grades, so there will be no grading process. All that you do you do for yourself. Everyone here knows what programming is & I advise you to write the best code you can.

So, let's dive right into it:

- [Rust homepage](#)

A dark green rectangular infographic titled "Why Rust?". It is divided into three columns, each with a heading and a paragraph of text.


Performance	Reliability	Productivity
Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.	Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.	Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

## *Rust's advantages*

- [Rust wiki page](#)



The official Rust logo

<b>Paradigms</b>	Multi-paradigm: concurrent, functional, generic, imperative, structured
<b>Designed by</b>	Graydon Hoare
<b>Developer</b>	The Rust Foundation
<b>First appeared</b>	July 7, 2010; 11 years ago
<b>Stable release</b>	1.61.0 <sup>[1]</sup>  / May 19, 2022; 7 days ago
<b>Typing discipline</b>	Affine, inferred, nominal, static, strong

*Rust wiki page description*

- [Rust vs C++](#)

Both Rust and C++:

- compile to native code,
- have no runtime,
- have no garbage collection,
- have direct access to memory (if it is needed),
- are low-level programming languages—they operate close to the hardware.

- [Steve Donovan's Rust intro](#)

## Hello, World!

The original purpose of "hello world", ever since the first C version was written, was to test the compiler and run an actual program.

```
// hello.rs
fn main() {
    println!("Hello, World!");
}
```

```
$ rustc hello.rs
$ ./hello
Hello, World!
```

*Hello, World!*

Rust is a curly-braces language with semicolons, C++-style comments and a `main` function - so far, so familiar. The exclamation mark indicates that this is a *macro* call. For C++ programmers, this can be a turn-off, since they are used to seriously stupid C macros - but I can ensure you that these macros are more capable and sane.

For anybody else, it's probably "Great, now I have to remember when to say bang!". However, the compiler is unusually helpful; if you leave out that exclamation, you get:

```
error[E0425]: unresolved name `println`
--> hello2.rs:2:5
  |
2 |     println("Hello, World!");
  |     ^^^^^^^ did you mean the macro `println!`?
```

*Compile error*

The next step is to introduce a *variable*:

```
// let1.rs
fn main() {
    let answer = 42;
    println!("Hello {}", answer);
}
```

## Variable

```
// for1.rs
fn main() {
    for i in 0..5 {
        println!("Hello {}", i);
    }
}
```

The *range* is not inclusive, so `i` goes from 0 to 4. This is convenient in a language which *indexes* things like arrays from 0.

And interesting things have to be done *conditionally*:

```
// for2.rs
fn main() {
    for i in 0..5 {
        if i % 2 == 0 {
            println!("even {}", i);
        } else {
            println!("odd {}", i);
        }
    }
}
```

```
even 0
odd 1
even 2
odd 3
even 4
```

## For-loop

This does the same, written in a more interesting way:

```
// for3.rs
fn main() {
    for i in 0..5 {
        let even_odd = if i % 2 == 0 {"even"} else {"odd"};
        println!("{}", even_odd, i);
    }
}
```

Traditionally, programming languages have *statements* (like `if`) and *expressions* (like `1+i`). In Rust, nearly everything has a value and can be an expression. The seriously ugly C 'ternary operator' `i % 2 == 0 ? "even" : "odd"` is not needed.

*Nearly everything is an expression*

## Adding Things Up

Computers are very good at arithmetic. Here is a first attempt at adding all the numbers from 0 to 4:

```
// add1.rs
fn main() {
    let sum = 0;
    for i in 0..5 {
        sum += i;
    }
    println!("sum is {}", sum);
}
```

But it fails to compile:

```
error[E0384]: re-assignment of immutable variable `sum`
--> add1.rs:5:9
3 |     let sum = 0;
  |     --- first assignment to `sum`
4 |     for i in 0..5 {
5 |         sum += i;
  |         ^^^^^^^ re-assignment of immutable variable
```

'Immutable'? A variable that cannot *vary*? `let` variables by default can only be assigned a value when declared. Adding the magic word `mut` (*please make this variable mutable*) does the trick:

```
// add2.rs
fn main() {
    let mut sum = 0;
    for i in 0..5 {
        sum += i;
    }
    println!("sum is {}", sum);
}
```

*Immutability*

## Function Types are Explicit

*Functions* are one place where the compiler will not work out types for you. And this in fact was a deliberate decision, since languages like Haskell have such powerful type inference that there are hardly any explicit type names. It's actually good Haskell style to put in explicit type signatures for functions. Rust requires this always.

Here is a simple user-defined function:

```
// fun1.rs

fn sqr(x: f64) -> f64 {
    return x * x;
}

fn main() {
    let res = sqr(2.0);
    println!("square is {}", res);
}
```

### *Functions*

A few more examples of this no-return expression style:

```
// absolute value of a floating-point number
fn abs(x: f64) -> f64 {
    if x > 0.0 {
        x
    } else {
        -x
    }
}

// ensure the number always falls in the given range
fn clamp(x: f64, x1: f64, x2: f64) -> f64 {
    if x < x1 {
        x1
    } else if x > x2 {
        x2
    } else {
        x
    }
}
```

It's not wrong to use `return`, but code is cleaner without it. You will still use `return` for *returning early* from a function.

### *How returns work*

Values can also be passed by *reference*. A reference is created by `&` and *dereferenced* by `*`.

```
fn by_ref(x: &i32) -> i32 {
    *x + 1
}

fn main() {
    let i = 10;
    let res1 = by_ref(&i);
    let res2 = by_ref(&41);
    println!("{}", res1, res2);
}
// 11 42
```

### Immutable references

What if you want a function to modify one of its arguments? Enter *mutable references*:

```
// fun4.rs

fn modifies(x: &mut f64) {
    *x = 1.0;
}

fn main() {
    let mut res = 0.0;
    modifies(&mut res);
    println!("res is {}", res);
}
```

This is more how C would do it than C++. You have to explicitly pass the reference (with `&`) and explicitly *dereference* with `*`. And then throw in `mut` because it's not the default. (I've always felt that C++ references are too easy to miss compared to C.)

Basically, Rust is introducing some *friction* here, and not-so-subtly pushing you towards returning values from functions directly. Fortunately, Rust has powerful ways to express things like "operation succeeded and here's the result" so `&mut` isn't needed that often. Passing by reference is important when we have a large object and don't wish to copy it.

The type-after-variable style applies to `let` as well, when you really want to nail down the type of a variable:

```
let bigint: i64 = 0;
```

### Mutable references

## Arrays and Slices

All statically-typed languages have *arrays*, which are values packed nose to tail in memory. Arrays are *indexed* from zero:

```
// array1.rs
fn main() {
    let arr = [10, 20, 30, 40];
    let first = arr[0];
    println!("first {}", first);

    for i in 0..4 {
        println!("[{}] = {}", i, arr[i]);
    }
    println!("length {}", arr.len());
}
```

And the output is:

```
first 10
[0] = 10
[1] = 20
[2] = 30
[3] = 40
length 4
```

### Arrays

More on arrays: [here](#)

- [The Book](#)

## What Is Ownership?

*Ownership* is a set of rules that governs how a Rust program manages memory. All programs have to 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. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running.

### Ownership intro



## Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate them:

- 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.

### *Ownership rules*

#### Ways Variables and Data Interact: Move

Multiple variables can interact with the same data in different ways in Rust. Let's look at an example using an integer in Listing 4-2.

```
let x = 5;  
let y = x;
```



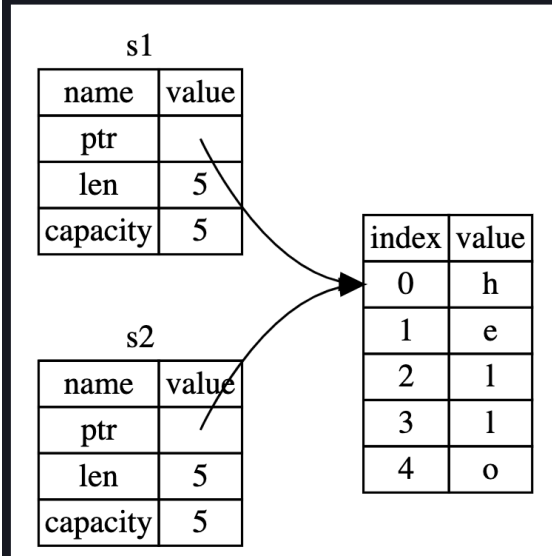
Listing 4-2: Assigning the integer value of variable `x` to `y`

We can probably guess what this is doing: “bind the value `5` to `x`; then make a copy of the value in `x` and bind it to `y`.” We now have two variables, `x` and `y`, and both equal `5`. This is indeed what is happening, because integers are simple values with a known, fixed size, and these two `5` values are pushed onto the stack.

### *Stack-allocated move semantics*

The length is how much memory, in bytes, the contents of the `String` is currently using. The capacity is the total amount of memory, in bytes, that the `String` has received from the allocator. The difference between length and capacity matters, but not in this context, so for now, it's fine to ignore the capacity.

When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to. In other words, the data representation in memory looks like Figure 4-2.



### *Heap-allocated move semantics*

So, as you can see, Rust is a general-purpose programming language, just like a multitude of other languages. At the next meeting, we are going to practice Rust with Rustlings - a fun & interactive way to learn Rust by correcting the code (just like RubyKoans).

### *Practice: Rustlings*

Hey everyone! So today is our first practice. During practices, we will mostly look at the code & practical aspects of things we learn during the lectures. I hope to see participation from you, but if not, I will continue with the answer. Today we are going to take a look at Rustlings. This is an interactive Rust learning app, which I have personally used and attribute a lot of credit to my Rust knowledge to it. We'll work on the problems in Rustlings <https://github.com/rust-lang/rustlings>

```
// intro1.rs
// About this `I AM NOT DONE` thing:
// We sometimes encourage you to keep trying things on a given exercise, even
// after you already figured it out. If you got everything working and feel
// ready for the next exercise, remove the `I AM NOT DONE` comment below.
// Execute the command `rustlings hint intro1` for a hint.

// I AM NOT DONE

fn main() {
    println!("Hello and");
    println!(r#"      welcome to...          "#);
    println!(r#"      _ _ _          "#);
    println!(r#" _ _ _ _ _ | | | ( _ ) _ _ _ _ _ "#);
    println!(r#" | ' _ | | / _ | _ | | ' _ \ / _ / _ | "#);
    println!(r#" | | | | | \ _ \ | | | | | | | ( _ | \ _ \ "#);
    println!(r#" | _ | \ _ , _ | _ \ _ | | | | | | | \ _ , | _ / "#);
    println!(r#"                      | _ /          "#);
    println!();
    println!("This exercise compiles successfully. The remaining exercises contain a compiler");
    println!("or logic error. The central concept behind Rustlings is to fix these errors and");
    println!("solve the exercises. Good luck!");
}
```

What should we add here to make the code compile?

```
// intro2.rs
// Make the code print a greeting to the world.
// Execute `rustlings hint intro2` for a hint.

// I AM NOT DONE

fn main() {
    println!("Hello {}!");
}
```

Let's check the formatting [information](#).

## Week 2

- Rust revision (advanced)
- Blockchain development
- Ethereum

Materials: [The Book](#), [What is bitcoin](#), [What is Web3](#), [Hyperledger Fabric](#)

**Homework:** Create a simple CLI signer for **ed25519** signatures. Possible libraries: clap, ed25519, log

### *Lecture: Advanced Rust*

Alright, since we need to move at a mad pace to complete the course, we'll try & cover everything we need in two lectures this week. The first lecture would be about higher-level Rust, while the second would be about blockchain development. We need to understand high-level Rust in order to be able to reflect blockchain concepts.

The high-level Rust we are going to talk about today is traits & generic mostly. Traits are the behavior inheritance mechanism in Rust. They are similar to interfaces in other languages (but not quite).

- [Traits](#)

## Traits: Defining Shared Behavior

A *trait* defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

Traits are collections of some data-agnostic behavior. That's really the bottom line.

We can define a trait the following way:

```
pub trait Summary {  
    fn summarize(&self) -> String;  
}
```



Let's look at how to implement a trait for 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)
    }
}

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, self.username, self.content)
    }
}
```

As we can see, we don't use the content field of NewsArticle. We don't need the contents in the summary.

Now, can the traits have a default implementation? Turns out they can! That is unlike in many other languages.

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Here, we don't use self as we cannot know, which fields the implementer has.

Though, we can control that in a way - let's take a look in the advanced traits section.

- [Advanced Traits](#)

Types in Rust might implement a lot of traits. It is not uncommon for some traits to have methods with equal names. Moreover, if a type has a method, say, foo, it's possible to implement a multitude of traits with method foo on this type.

“When calling methods with the same name, you'll need to tell Rust which one you want to use. Consider the code in Listing 19-16 where we've defined two traits, Pilot and Wizard, that both have a method called fly. We then implement both traits on a type Human that already has a method named fly implemented on it. Each fly method does something

different.” (<https://doc.rust-lang.org/book/ch19-03-advanced-traits.html>)

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

So, how do we differentiate between those different methods? How do we call them?

Question: What will this code do?

```
fn main() {
    let person = Human;
    person.fly();
}
```

Below we show how to call the methods of different traits on some value.

To call the `fly` methods from either the `Pilot` trait or the `Wizard` trait, we need to use more explicit syntax to specify which `fly` method we mean. Listing 19-18 demonstrates this syntax.

Filename: `src/main.rs`

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-18: Specifying which trait's `fly` method we want to call

Running this code prints the following:

```
$ cargo run
  Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
    Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

“However, associated functions that are not methods don’t have a self parameter. When there are multiple types or traits that define non-method functions with the same function name, Rust doesn’t always know which type you mean unless you use fully qualified syntax. For example, the `Animal` trait in Listing 19-19 has the associated non-method function `baby_name`, and the `Animal` trait is implemented for the struct `Dog`. There’s also an associated non-method function `baby_name` defined on `Dog` directly.”

(<https://doc.rust-lang.org/book/ch19-03-advanced-traits.html>)

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

In `main`, we call the `Dog::baby_name` function, which calls the associated function defined on `Dog` directly. This code prints the following:

```
$ cargo run
  Compiling traits-example v0.1.0 (file:///projects/traits-example)
  Finished dev [unoptimized + debuginfo] target(s) in 0.54s
  Running `target/debug/traits-example`
A baby dog is called a Spot
```

This output isn't what we wanted. We want to call the `baby_name` function that is part of the `Animal` trait that we implemented on `Dog` so the code prints `A baby dog is called a puppy`. The technique of specifying the trait name that we used in Listing 19-18 doesn't help here; if we change `main` to the code in Listing 19-20, we'll get a compilation error.

Filename: `src/main.rs`

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

Generally, to disambiguate the type, we need to use the following syntax: `<Type as Trait>::method(args)`

Here's an example that refers to our `Animal` hierarchy:

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

heavily used in `Substrate`, as we will see later.

This is

There's also one more thing we need to know: supertraits and trait bounds. Let's look at another example from the Book.

For example, let's say we want to make an `OutlinePrint` trait with an `outline_print` method that will print a value framed in asterisks. That is, given a `Point` struct that implements `Display` to result in `(x, y)`, when we call `outline_print` on a `Point` instance that has `1` for `x` and `3` for `y`, it should print the following:

```
*****
*      *
* (1, 3) *
*      *
*****
```



```
use std::fmt;
```

```
trait OutlinePrint: fmt::Display {  
    fn outline_print(&self) {  
        let output = self.to_string();  
        let len = output.len();  
        println!("{}", "*".repeat(len + 4));  
        println!("*{}*", " ".repeat(len + 2));  
        println!("* {} *", output);  
        println!("*{}*", " ".repeat(len + 2));  
        println!("{}", "*".repeat(len + 4));  
    }  
}
```

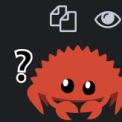


Because we've specified that `OutlinePrint` requires the `Display` trait, we can use the `to_string` function that is automatically implemented for any type that implements `Display`. If we tried to use `to_string` without adding a colon and specifying the `Display` trait after the trait name, we'd get an error saying that no method named `to_string` was found for the type `&Self` in the current scope.

Let's see what happens when we try to implement `OutlinePrint` on a type that doesn't implement `Display`, such as the `Point` struct:

Filename: src/main.rs

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl OutlinePrint for Point {}
```



We get an error saying that `Display` is required but not implemented:

```
$ cargo run  
Compiling traits-example v0.1.0 (file:///projects/traits-example)  
error[E0277]: `Point` doesn't implement `std::fmt::Display`  
--> src/main.rs:20:6  
|  
20 | impl OutlinePrint for Point {}  
|    ^^^^^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter  
|  
= help: the trait `std::fmt::Display` is not implemented for `Point`  
= note: in format strings you may be able to use `{:?}` (or `{:#?}` for pretty)  
note: required by a bound in `OutlinePrint`  
--> src/main.rs:3:21  
|  
3 | trait OutlinePrint: fmt::Display {  
|    ^^^^^^^^^^^^^^^^^ required by this bound in `OutlinePrint`  
  
For more information about this error, try `rustc --explain E0277`.  
error: could not compile `traits-example` due to previous error
```

Let's take a look at the complete, working example.

```

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}

struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}

use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

fn main() {
    let p = Point { x: 1, y: 3 };
    p.outline_print();
}

```

Now, as we got traits fully out of the way, let's dive into macros. It might seem intimidating at the first glance, but I promise they'll make sense in the end!

- [Macros](#)

So, in short, macros are a compile-time code generation tool. They are very different from the macros we are used to in C and C++: these macros are hygienic. You can read more about macro hygiene [here](#).

We've used macros like `println!` throughout this book, but we haven't fully explored what a macro is and how it works. The term *macro* refers to a family of features in Rust: *declarative* macros with `macro_rules!` and three kinds of *procedural* macros:

- Custom `#[derive]` macros that specify code added with the `derive` attribute used on structs and enums
- Attribute-like macros that define custom attributes usable on any item
- Function-like macros that look like function calls but operate on the tokens specified as their argument

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

This example shows us the `vec!` macro. We can use `vec` to conveniently create vectors with some elements (like with an initializer in `c++`):

```
fn main() {
    let my_vec = vec![1, 2, 3, 4, 5];
    my_vec.into_iter().for_each(|item| println!("{}", item))
    // this is a new syntax for printing captured vars,
    // btw
}
```

So, in the macro definition, we can see some weird syntax with dollar signs, types, stars etc. In our scope, we will not look into how macros work, but in general, those are syntactic constructs that define some calling syntax and output some code on invocation (during compilation).

So in our case, the `vec!` invocation will turn into this:

```
fn main() {
    let my_vec = {
        let mut temp_vec = Vec::new();
        temp_vec.push(1);
        temp_vec.push(2);
        temp_vec.push(3);
        temp_vec.push(4);
        temp_vec.push(5);
        temp_vec
    };
}
```

```
    my_vec.into_iter().for_each(|item| println!("{item}"))
}
```

Another type of macros is procedural macros. They look like this: `#[some_attribute]`.

We won't look much into them, only how to use them. The general difference between procedural & declarative macros is that procedural macros are full-blown programs running in the compiler, while declarative macros are kinda like functions for code generation. Procedural macros operate on `TokenStreams`, while their declarative counterparts operate on concrete language constructs: expressions, literals, types, visibility items, and other stuff.

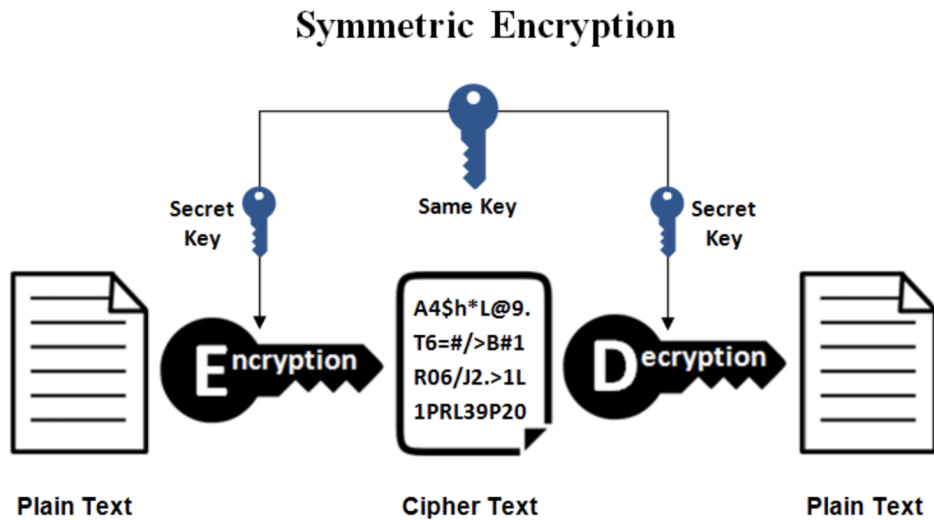
Today we've taken a look at the most important features of advanced Rust used in Substrate. That's on top of *all* of the basic features, of course. This is the end of our Rust study/revision. Next, we will have some theoretical stuff to work through, but then we'll get to writing Substrate. This will be one big practice for Rust, as well as blockchain development concepts we'll learn about next.

## *Lecture 2*

Hello again, everyone! Today is the second lecture of our week. In this lecture, we'll be talking about blockchains in general. We'll work using a [blockchain learning path](#) kindly provided by Kyrlo Gorokhovskyy to me. We'll cover the basics as much as possible.

So, let's start with simple things: primitives of blockchain technology.

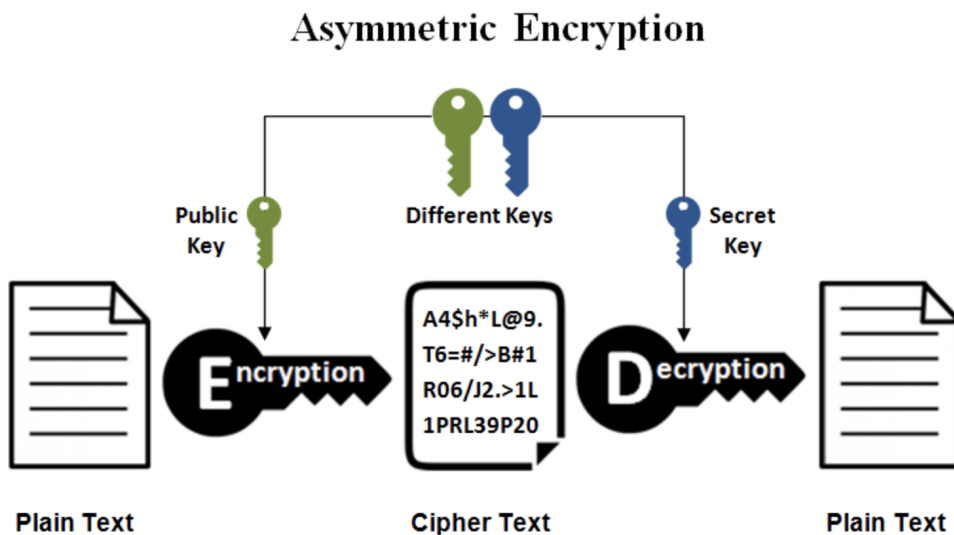
## What is Symmetric Encryption?



This is the simplest kind of encryption that involves only one secret key to cipher and decipher information. Symmetric encryption is an old and best-known technique. It uses a secret key that can either be a number, a word or a string of random letters. It is a blended with the plain text of a message to change the content in a particular way. The sender and the recipient should know the secret key that is used to encrypt and decrypt all the messages. Blowfish, AES, RC4, DES, RC5, and RC6 are examples of symmetric encryption. The most widely used symmetric algorithm is AES-128, AES-192, and AES-256.

The main disadvantage of the symmetric key encryption is that all parties involved have to exchange the key used to encrypt the data before they can decrypt it.

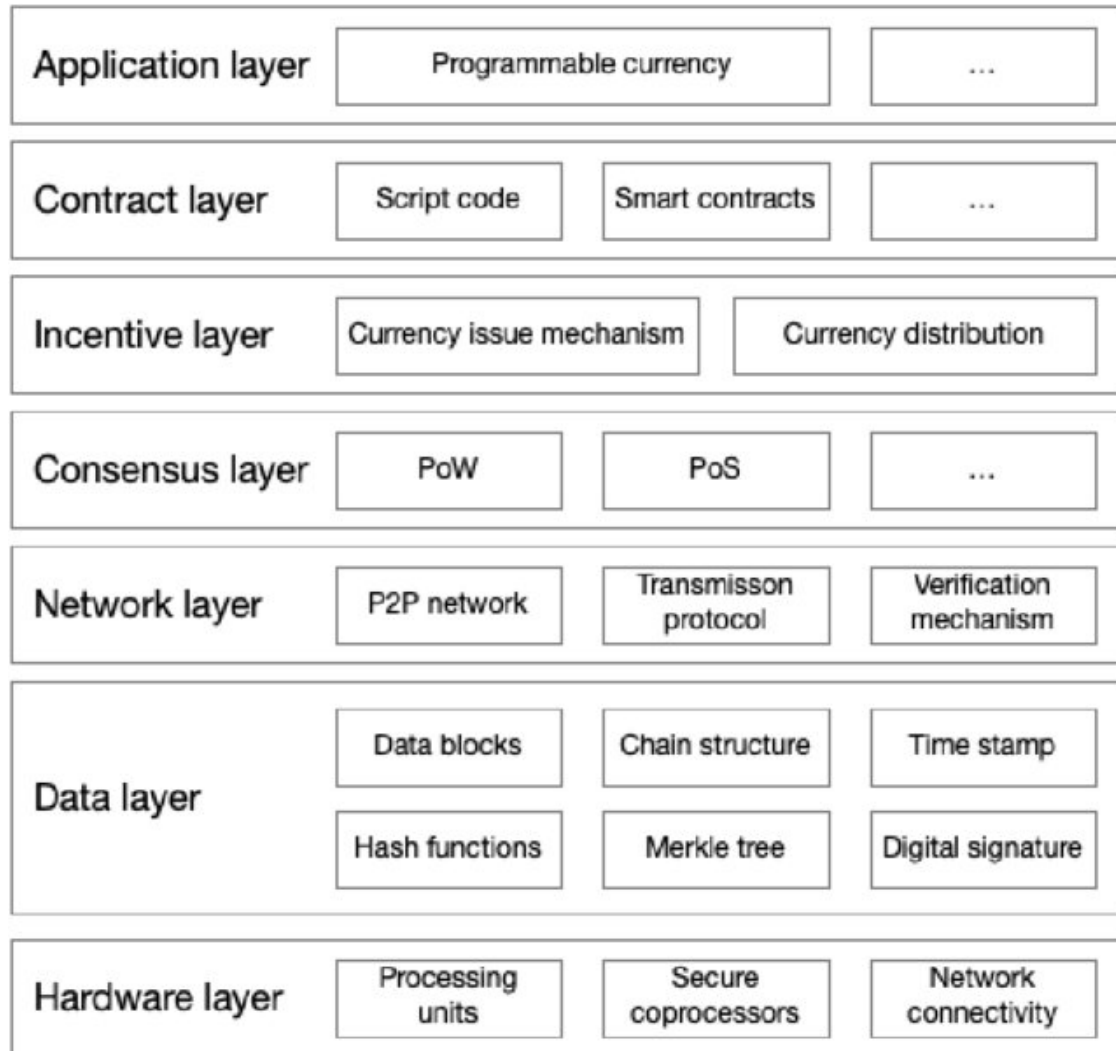
## What is Asymmetric Encryption?



After we got the crypto intricacies out of the way, let's check how bitcoin works. This will take some time.

After covering the most basic stuff (hashes, digital signatures, cryptography, PKI, and so on), we will move on to covering [bitcoin](#). Bitcoin is the first cryptocurrency, but it is really advanced. Definitely worth understanding how it works.

[This](#) is the anatomy of the node:



After getting an understanding of how Bitcoin works, let's take a look at its consensus algorithm: [Proof of Work](#). One great [video](#) on how bitcoin works.

### [Week 3](#)

**Homework:** go through all of the materials, and finish the Ethereum article.

*[Lecture: Ethereum, smart contracts](#)*

Understanding the first advanced smart contract platform, Ethereum, is essential to understanding the whole blockchain development fuss. Ethereum was the first blockchain to show that one can write decentralized programs, which would be verified by all network participants.

You might ask, why they are called contracts. Well, to be fair, I don't know. But I'm pretty sure it was some gimmick to attract business people to adapt Ethereum. Anyways.

Today we'll spend most of the lecture revising how Ethereum works, so prepare yourself for a pretty big journey [here](#).

Let's look at the blockchain definition from the article:

### **Blockchain definition**

A blockchain is a “**cryptographically secure transactional singleton machine with shared-state.**” [1] That's a mouthful, isn't it? Let's break it down.

- “**Cryptographically secure**” means that the creation of digital currency is secured by complex mathematical algorithms that are obscenely hard to break. Think of a firewall of sorts. They make it nearly impossible to cheat the system (e.g. create fake transactions, erase transactions, etc.)
- “**Transactional singleton machine**” means that there's a single canonical instance of the machine responsible for all the transactions being created in the system. In other words, there's a single global truth that everyone believes in.
- “**With shared-state**” means that the state stored on this machine is shared and open to everyone.

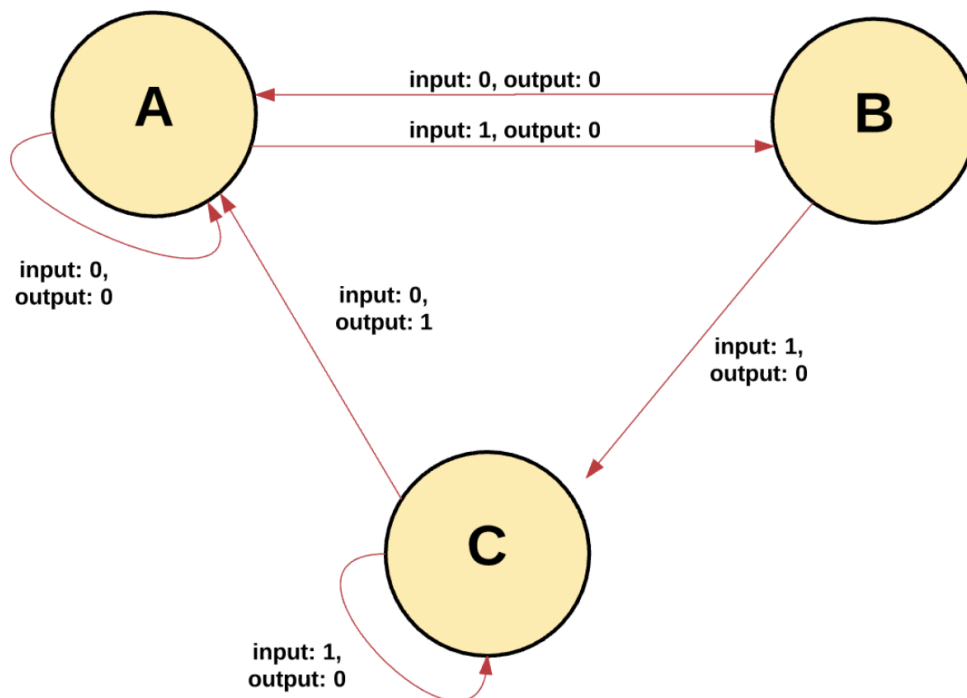
Ethereum implements this blockchain paradigm.



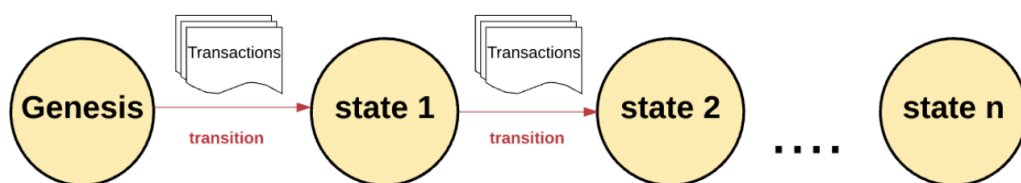
A more detailed explanation of Ethereum's paradigm:

### The Ethereum blockchain paradigm explained

The Ethereum blockchain is essentially a **transaction-based state machine**. In computer science, a *state machine* refers to something that will read a series of inputs and, based on those inputs, will transition to a new state.



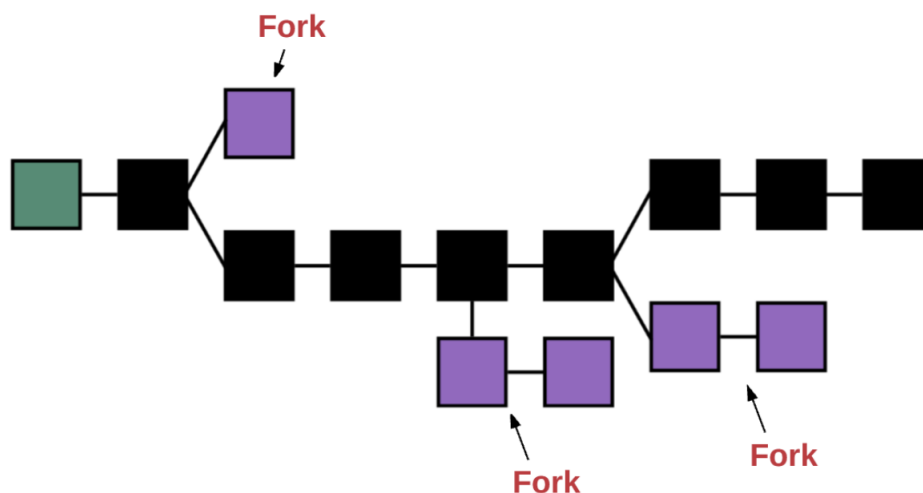
With Ethereum's state machine, we begin with a "genesis state." This is analogous to a blank slate, before any transactions have happened on the network. When transactions are executed, this genesis state transitions into some final state. At any point in time, this final state represents the current state of Ethereum.



Here we can see a diagram of how the state progresses within the blockchain. The possible state changes between some block N and block N+1 can be described by a state transition function. The state transition function is the whole collection of things that can change the chain state - smart contract calls, logs, balance transfers, paying fees etc. In other words, we can define everything that happens on a blockchain by its state transition function. This is a common thing for all blockchains that contain any logic - even bitcoin. Bitcoin script executions, UTXO assignment - those are the parts of the state transition function of bitcoin. STF is an important term to remember, as Substrate also has its state transition function. In Substrate, the state transition function is fully customizable and is named runtime.

Earlier, we defined a blockchain as a **transactional singleton machine with shared-state**. Using this definition, we can understand the correct current state is a single global truth, which everyone must accept. Having multiple states (or chains) would ruin the whole system, because it would be impossible to agree on which state was the correct one. If the chains were to diverge, you might own 10 coins on one chain, 20 on another, and 40 on another. In this scenario, there would be no way to determine which chain was the most “valid.”

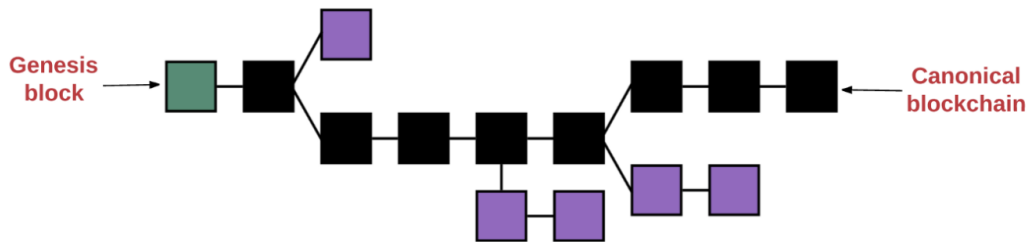
Whenever multiple paths are generated, a “fork” occurs. We typically want to avoid forks, because they disrupt the system and force people to choose which chain they “believe” in.



This diagram shows us the results of the process of block production. When we produce blocks, we might naturally get forks, because different validator nodes might be on different blocks when producing the next block. To choose the correct block, we need some mechanism. This mechanism is different from blockchain to blockchain, but since we’re

talking about Ethereum now, let's look at GHOST.

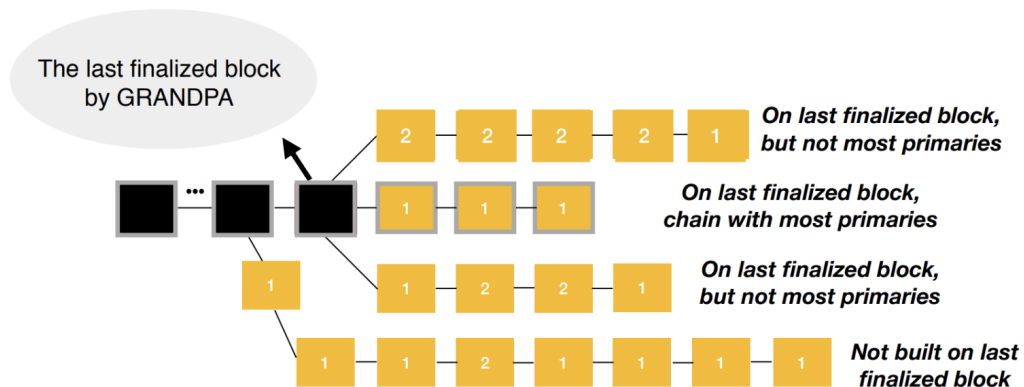
In simple terms, **the GHOST protocol says we must pick the path that has had the most computation done upon it.** One way to determine that path is to use the block number of the most recent block (the “leaf block”), which represents the total number of blocks in the current path (not counting the genesis block). The higher the block number, the longer the path and the greater the mining effort that must have gone into arriving at the leaf. Using this reasoning allows us to agree on the canonical version of the current state.



Basically, we deem the blocks which base on the longest finalized chain as correct. Polkadot has something similar:

## Fork Choice

Bringing BABE and GRANDPA together, the fork choice of Polkadot becomes clear. BABE must always build on the chain that has been finalized by GRANDPA. When there are forks after the finalized head, BABE provides probabilistic finality by building on the chain with the most primary blocks.



Longest chain with most primaries on last finalized GRANDPA block

In the above image, the black blocks are finalized, and the yellow blocks are not. Blocks marked with a "1" are primary blocks; those marked with a "2" are secondary blocks. Even though the topmost chain is the longest chain on the latest finalized block, it does not qualify because it has fewer primaries at the time of evaluation than the one below it.

We're not going to get much deeper than that. This understanding of finality should be enough.

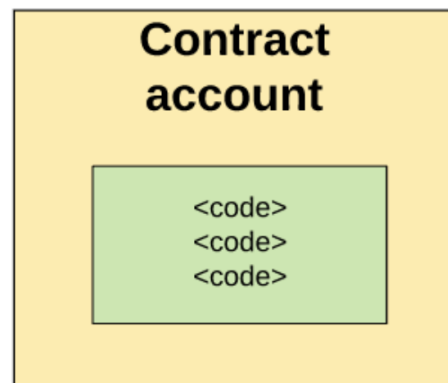
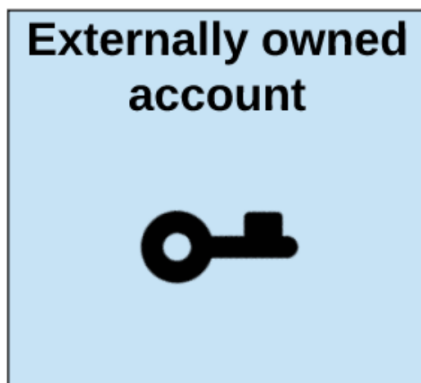
Now, let's look at the account model of Ethereum:

## Accounts

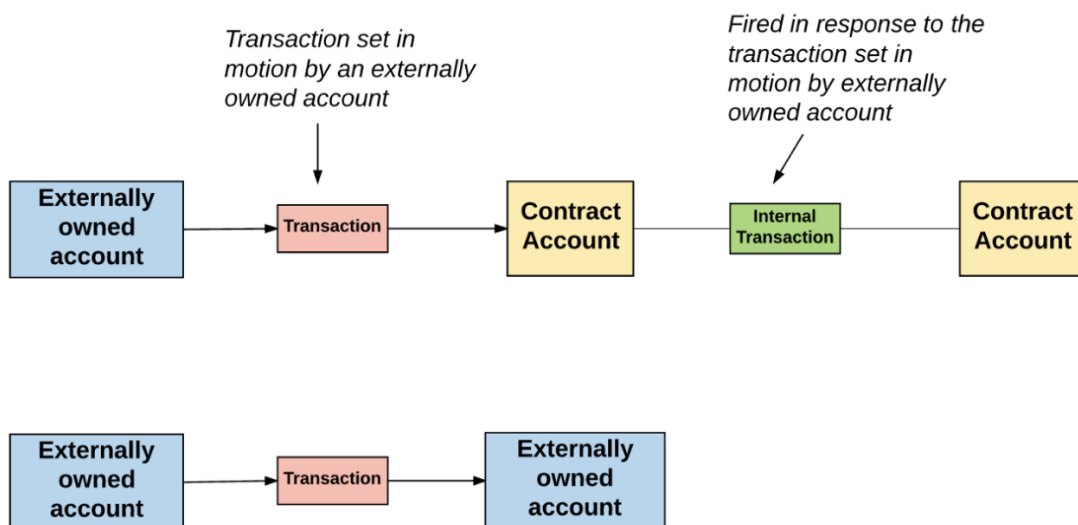
The global “shared-state” of Ethereum is comprised of many small objects (“accounts”) that are able to interact with one another through a message-passing framework. Each account has a **state** associated with it and a 20-byte **address**. An address in Ethereum is a 160-bit identifier that is used to identify any account.

There are two types of accounts:

- Externally owned accounts, which are **controlled by private keys** and have **no code associated with them**.
- Contract accounts, which are **controlled by their contract code** and **have code associated with them**.

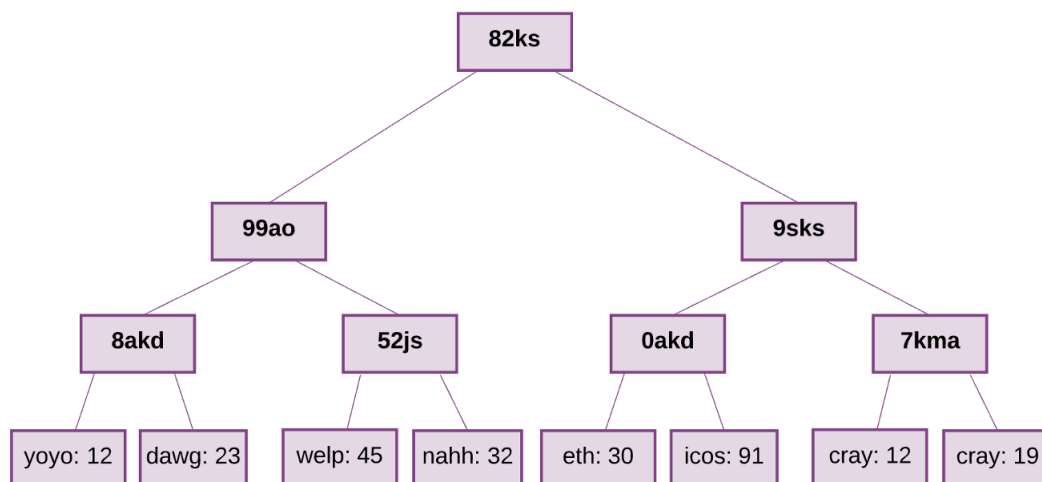


Below we can see a diagram of how transactions are initiated on Ethereum:



Now, let's take a look at another important, defining thing in Ethereum & in account-based blockchains as a whole. The state of the blockchain is described by the ledger (connected blocks), but we also have the state, *which is attached to each account*. This state contains the transactions initiated by this account, the balances, etc. For contracts, this state also contains the code hash of the contract.

The state is described using Merkle Patricia trees, analogously called *tries*.



Let's take a deeper look into the article and talk about **gas** and **smart contracts**.

If we still have time after taking on this beast of an article, well, we can have some QnA time to discuss everything covered before.

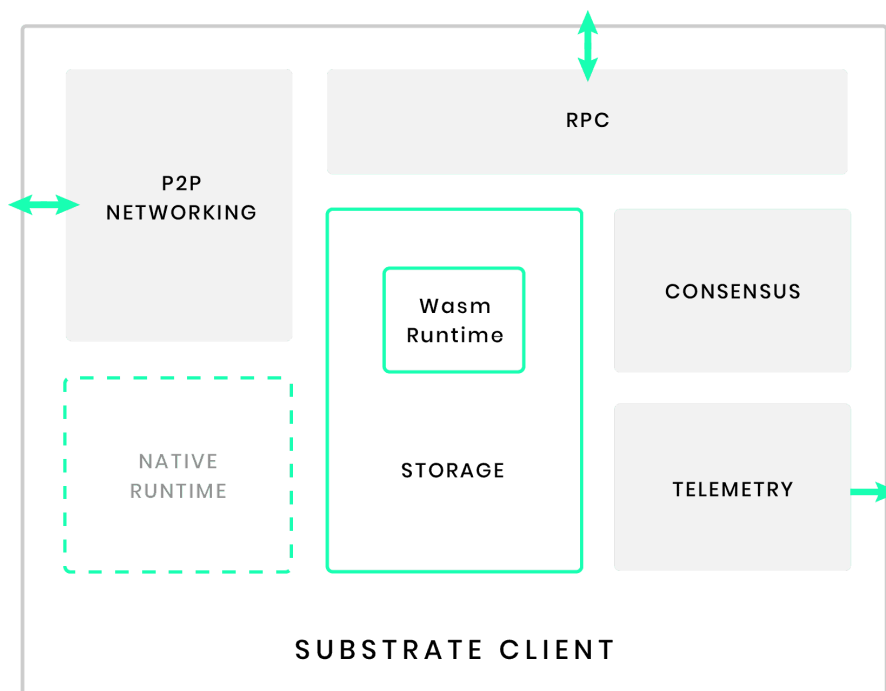
## Lecture: Substrate

Hello again, everyone! Today is the day we finally start learning Substrate. We've covered a lot in the previous two weeks, so let's use our knowledge in our main learning subject - Substrate.

Substrate is a framework that allows us to build efficient, future-proof, and modular blockchains with a LOT of features provided out-of-the-box.

Let's take a look at the general architecture of Substrate. What provides the flexibility of Substrate? The answer is FRAME. This architectural part is responsible for modularizing functionality, allowing for hot swaps & runtime code upgrades.

- General node architecture



This is the architecture of a substrate client (read *substrate node*). As we can see, it has a number of things in it:

- 1) P2P Networking module
- 2) RPC module
- 3) Storage with the WASM runtime in it
- 4) Consensus module

5) Telemetry module

6) Native runtime

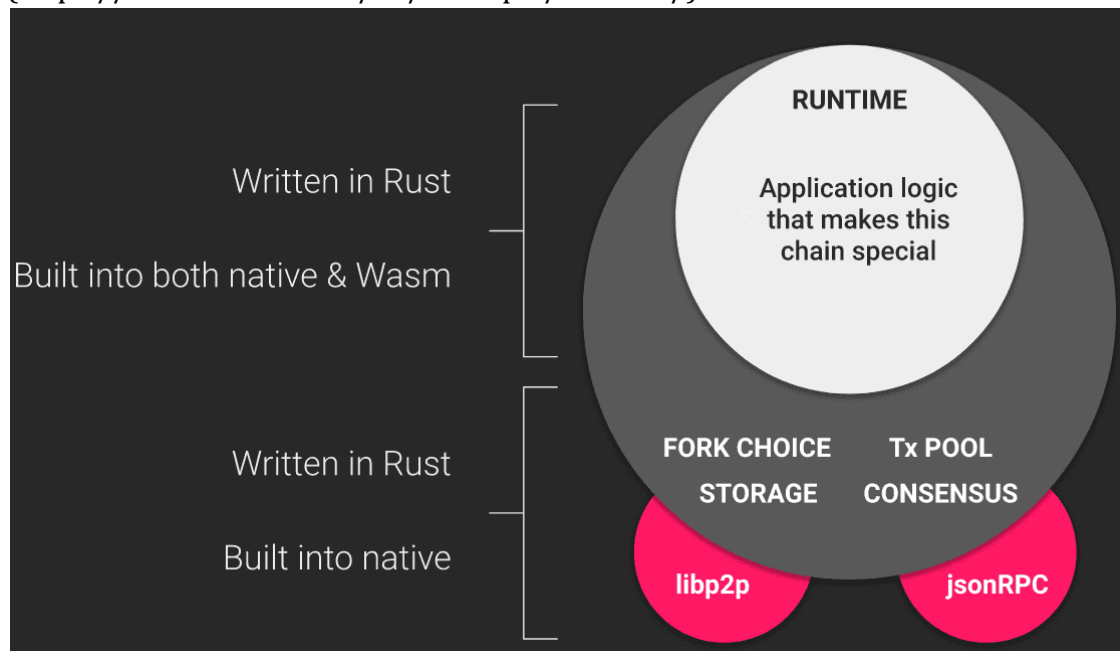
Let's quickly take a look one by one:

- The P2P networking module is responsible for transaction propagation. When a node receives a transaction via an RPC call, the node must propagate this transaction to the peers so that the whole network executes it.
- RPC is responsible for receiving the transaction calls & hosting different host function endpoints (as in chain-specific functions).
- Consensus is the module that is responsible for, you guessed it, consensus. The consensus is pluggable, meaning we can choose different consensus algorithms for different chains.
- Telemetry is just some endpoints that transmit node metrics.
- Native runtime is everything that's available for use in and outside of the WASM runtime: same and more functions, same and more modules. The native runtime was created to overcome the slowness of the WASM runtime, but it cannot provide guarantees of equal execution of STF between different nodes. Simply, if you're gonna expect equal computation results of equal calls on different nodes, you might not get them.
- [Runtime architecture](#)

“The runtime of a blockchain is the business logic that defines its behavior. In Substrate-based chains, the runtime is referred to as the "state transition function"; it is where Substrate developers define the storage items that are used to represent the blockchain's state as well as the functions that allow blockchain users to make changes to this state.”



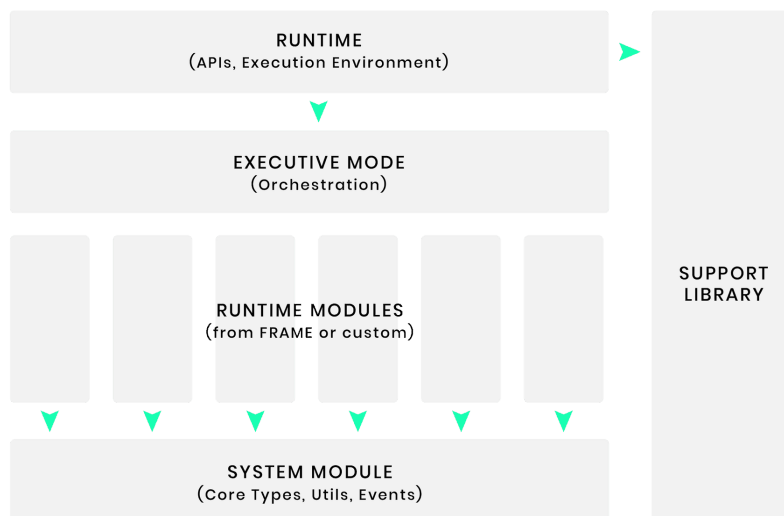
(<https://docs.substrate.io/v3/concepts/runtime/>)



As we can see, everything that makes a chain special in terms of its application logic is compiled to WASM and stored in the storage. Everything else is either part of each individual node (tx pool, finality, storage, block production, RPC, p2p server, native runtime).

- **FRAME**

FRAME is an architectural approach to building a substrate node, which makes the chain flexible and easily upgradeable. FRAME stands for the *Framework for Runtime Aggregation of Modularized Entities*. That's a mouthful.



Looking

at this diagram, we see the following: - Runtime. Runtime is composed of default/custom FRAME *pallets*, execution module, support library, and the system module. - The Executive is responsible for executing the transactions within a block, progressing block state, and other core block-related things. - Runtime modules fill our runtime with logic - we can add the Balances pallet to track the balances of an account, the Transaction Payment pallet to enable the payments for the transaction execution, the Timestamp pallet to insert block timestamps, etc. - The System module is mainly responsible for managing how our runtime is constructed, managing transaction origins, etc. - The Support Library is just a collection of useful *traits*, *macros* and other stuff related to our blockchain.

Out of all those things, those are the Runtime Modules we choose that make our runtime special.

Now, let's talk about the transactions, cause what is a blockchain without transactions?!

- [Extrinsics](#)

Let's take a look at the definition of an *extrinsic*:

An extrinsic is a piece of information that comes from outside the chain and is included in a block. Extrinsics fall into three categories: inherents, signed transactions, and unsigned transactions.

So, how is it different from transactions? Answer: A block in Substrate is composed of a header and an array of extrinsics. The header contains a block height, parent hash, extrinsics root, state root, and digest.

Basically, extrinsics are very similar to transactions. The difference is rather in the etymology of the word itself - *transactions* are thought of as something that has a monetary value (or execute some logic in the case of blockchains), while *extrinsics* are more generally pieces of external information.

Frankly, we, Substrate devs, use *transactions* and *extrinsic* interchangeably.

Let's talk about signed and unsigned transactions. The difference is mostly inferred from the name. Signed transactions are signed by network participants (as in people), while unsigned txs are not signed by anyone. This means we have to define custom validation logic for those transactions. Signed transactions are like Ethereum transactions, while unsigned transactions can be thought of more like system-related transactions. One must use it with care because since no one signs those txs, no one pays the fee.

- [Weights](#)

When we use our blockchain, the gas is used to provide to miners for the following:

- Memory usage
- Storage input and output
- Computation
- Transaction and block size
- State database size

Substrate does not use the concept of **gas**, as it doesn't have predefined opcodes like the smart contracts in Ethereum. Instead, we pay based on the length of our tx, state changes & **the time consumed for the computation**. You might say that on each node the execution times might be different, and you would be right. That's why Substrate standardizes hardware, based on which the weights are estimated.

Substrate defines one unit of weight as one picosecond of execution time, that is  $10^{12}$  weight = 1 second, or 1,000 weight = 1 nanosecond, on fixed reference hardware (Intel Core i7-7700K CPU with 64GB of RAM and an NVMe SSD).

Benchmarking on reference hardware makes weights comparable across runtimes, which allows composability of software components from different sources. In order to tune a runtime for different validator hardware assumptions, you can set a different maximum block weight. For example, in order to allow validators to participate that are only half as fast as the reference machine, the maximum block weight should be half of the default, keeping the default block time.

## How fees are calculated

The final fee for a transaction is calculated using the following parameters:

- *base fee*: This is the minimum amount a user pays for a transaction. It is declared as a **base weight** in the runtime and converted to a fee using `WeightToFee`.
- *weight fee*: A fee proportional to the execution time (input and output and computation) that a transaction consumes.
- *length fee*: A fee proportional to the encoded length of the transaction.
- *tip*: An optional tip to increase the priority of the transaction, giving it a higher chance to be included by the transaction queue.

The base fee and proportional weight and length fees constitute the **inclusion fee**. The inclusion fee is the minimum fee that must be available for a transaction to be included in a block.

```
inclusion_fee = base_fee + length_fee + [targeted_fee_adjustment * weight_fee];  
final_fee = inclusion_fee + tip;
```

Materials: [Substrate docs](#), [Polkadot wiki](#)

### Week 4

**Homework:** Create a faucet that mints tokens to the requestor every, say, 16 blocks, meaning the requestor cannot get tokens if 16 blocks had not passed since the last mint. Also, since this transaction is lightweight and cannot be repeated more than 1 time in 16 blocks, **disable** transaction fees for that. The transaction must be signed. You can do the minting using the Currency trait and setting it to pallet\_balances' implementation (Balances in runtime/lib.rs). Use issue and resolve\_into\_existing methods. The implementation is totally up to you (minting right away, approving mints by superuser/governance - how far you want to go). More on that: [tight pallet coupling](#), [Currency trait](#).


### *Practice: Substrate FRAME*

Alright, so here comes our practice. During the previous lecture, we discussed how Substrate generally works, while also checking out Extrinsic & Weights. That's the most important info we should know to get right into Substrate development.

During this practice, we will complete [this](#) tutorial.

Let's dive right into this. Before we start with the code, I'd like to go over some theoretical aspects of a blockchain node, which are provided in the tutorial.

At a high level, a blockchain node consists of the following key components:

- [Storage](#)
- [Peer-to-peer networking](#) 
- [Consensus capabilities](#)
- Data handling capabilities for external or "[extrinsic](#)" information
- A [Runtime](#)

So, as we have seen before, a blockchain node consists of a couple of things. The **storage** is responsible for storing all of the accounts / UTXO data, the code to run, blockchain metadata & so on. Some nodes have a more extensive storage use (like Substrate, which uses storage for all of the STF code), and some have a less extensive storage use (like Bitcoin). The **P2P** networking is an essential part of a blockchain node since it is responsible for propagating the transactions among the node's peers and receiving such [gossip](#). The **consensus** says everything for itself. These **data handling capabilities** imply having some RPC/REST set up to receive the transactions, which are ultimately this **extrinsic information**. A **runtime** is that thing we talked about in the last lecture.

Now, let's go to the page of this tutorial & complete it.

(If we have time, we might take on [Nicks pallet tutorial](#)).

### *[Practice: runtime upgrades & governance](#)*

Hey! Getting familiar with Substrate already? Well, this week you'll become even **more** familiar with it. Today we are talking about more core features of Substrate that make it a great framework. And the first one I'd like to talk about is **forkless runtime upgrades**. This is a technique to update the runtime (remember, the state transition function) without actually redeploying the nodes. This is possible thanks to a WASM runtime. Since the code is just some bytes stored within the node's storage under some key, we can easily update it, just like we update values in a database. Of course, not all network participants can arbitrarily update the code for all nodes. This requires some higher privileges. While it is possible to do using pallet-sudo, having one privileged user within a decentralized network isn't typically a best practice. Instead, production chains (Polkadot, Kusama, Moonbeam, Acala, etc.) use various governance mechanisms to vote for an upgrade and update the code collectively. In this way, this DAO-based approach pretty much reflects modern governmental systems, with elections, voting, etc.

So, without further ado, let's get to it!

### [Forkless Upgrade tutorial](#)

To summarize, we checked out the Scheduler pallet & how to do forkless runtime upgrades.

Now, to do proper upgrades, we'll have to use some decentralized approach for deciding on the feasibility, security, and importance of an upgrade. pallet-collective will help us with that. To wrap things up, I'd like to check out the governance, presented on Polkadot & implemented using Substrate. [here](#).

## Week 5

### *Lecture - ink! and smart contracts*

Hello everyone! Today is the beginning of week 4 of our course. Today's topic is **ink! and smart contracts**.

ink! and smart contracts in Substrate are definitely **not** the most popular thing in the ecosystem. Although it's rigorously discussed within the community, no parachain (as of May 2022) supports WASM-based smart contracts in production (as in Polkadot, Kusama, etc.).

Then why would we take a look at it?! Well, ultimately, it's because we become better programmers when we learn from different languages, frameworks, etc. Moreover, we start to better understand the intrinsics of a system. In our case, we might compare Solidity/Near SDK/Solana Programs/etc to ink! to see commonalities and differences.

Let's dive right into it. ink! is an eDSL for pallet-contracts-based smart contracts. ink! is compiled to WASM. So, in other words, ink! is a language in which we write smart contracts (mind you, ink! is also valid Rust), then it compiles to WASM, then this WASM code gets executed in pallet-contracts. pallet-contracts is a stack-based virtual machine that supports WASM evaluation, memory, etc. As opposed to Solidity, ink! is not compiled to bytecode, which means it does not have its set of opcodes to operate with. Consequently, the concept of "gas" is very different in pallet-contracts rather than EVM.

With this general info, we might proceed & take a look at the practical part [here](#)!

Materials: [ink! docs](#), [Openbrush](#), [ink! tutorial](#)

**Homework:** Create a mintable-burnable PSP22 token using Openbrush (no UI integration needed).

### *QnA - Integrating a real-world application with blockchain & QnA with Gautam Dhameja*

Materials: [How to integrate a blockchain with a business](#)

So, this will be the last meeting in this course. We already covered a lot, had massive amounts of practical tasks, and learned a bit about the usage of Substrate. Today I'd like to invite Gautam Dhameja, Director of Solution Delivery @ Parity Technologies.

So, what I'd like to go through:

1. **Parachains, parathreads, relay chains - how to go from a Substrate chain running locally to the one real people are using**

2. **What is the benefit of using Polkadot, as opposed to say Ethereumv2**
3. **Smart contracts vs custom blockchain**
4. **How to build a business model around a Substrate-based blockchain**

## **Our logical conclusion**

This course has come to an end. We've covered a lot. I am sure that those who attended every meeting have learned something new, something that widens the outlook for the developer inside. I will always be happy to see you as builders, somewhere in the Web3 space, or just quietly enjoying Rust, even as a fun activity.

## **Literature used**

1. Rust homepage [Электронный ресурс] – Режим доступа до ресурсу:  
<https://www.rust-lang.org/>

2. Teams who build in Substrate [Електронний ресурс] – Режим доступу до ресурсу:  
<https://substrate.io/ecosystem/projects/>
3. Ukrainian Wikipedia Rust page [Електронний ресурс] – Режим доступу до ресурсу:  
[https://uk.wikipedia.org/wiki/Rust\\_\(%D0%BC%D0%BE%D0%B2%D0%B0\\_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F\)](https://uk.wikipedia.org/wiki/Rust_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F))
4. Rust vs C++ [Електронний ресурс] – Режим доступу до ресурсу:  
<https://codilime.com/blog/rust-vs-cpp-the-main-differences-between-these-popular-programming-languages/>
5. Steve Donovan's Rust intro [Електронний ресурс] – Режим доступу до ресурсу:  
<https://stevedonovan.github.io/rust-gentle-intro/1-basics.html>
6. The Rust Programming Language [Електронний ресурс] – Режим доступу до ресурсу:  
<https://doc.rust-lang.org/nightly/book/title-page.html>
7. Rustlings [Електронний ресурс] – Режим доступу до ресурсу:  
<https://www.rust-lang.org/>
8. Symmetric vs Assymmetric encryption [Електронний ресурс] – Режим доступу до ресурсу:  
<https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences>
9. What is bitcoin [Електронний ресурс] – Режим доступу до ресурсу:  
<https://academy.binance.com/uk/articles/what-is-bitcoin>
10. Blockchain in Cyberdefence: A Technology Review from a Swiss Perspective [Електронний ресурс] – Режим доступу до ресурсу:  
[https://www.researchgate.net/publication/349787845\\_Blockchain\\_in\\_Cyberdefence\\_A\\_Technology\\_Review\\_from\\_a\\_Swiss\\_Perspective/figures?lo=1](https://www.researchgate.net/publication/349787845_Blockchain_in_Cyberdefence_A_Technology_Review_from_a_Swiss_Perspective/figures?lo=1)
11. What is Proof of Work [Електронний ресурс] – Режим доступу до ресурсу:  
<https://academy.binance.com/uk/articles/proof-of-work-explained>
12. How does Ethereum work, anyway? [Електронний ресурс] – Режим доступу до ресурсу:  
<https://preethikasireddy.medium.com/how-does-ethereum-work-anyway-22d1df506369>



13. GRANDPA: A Byzantine Finality Gadget [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/w3f/consensus/blob/master/pdf/grandpa.pdf>
14. Substrate Documentation [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.substrate.io/v3/getting-started/overview/>
15. Polkadot wiki [Электронный ресурс] – Режим доступа до ресурсу: <https://wiki.polkadot.network/docs/getting-started>
16. Gossip Protocol [Электронный ресурс] – Режим доступа до ресурсу: <https://academy.binance.com/en/glossary/gossip-protocol>
17. Substrate Tutorials [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.substrate.io/tutorials/v3/>