

# Lesson 5

This week

Monday : Cairo

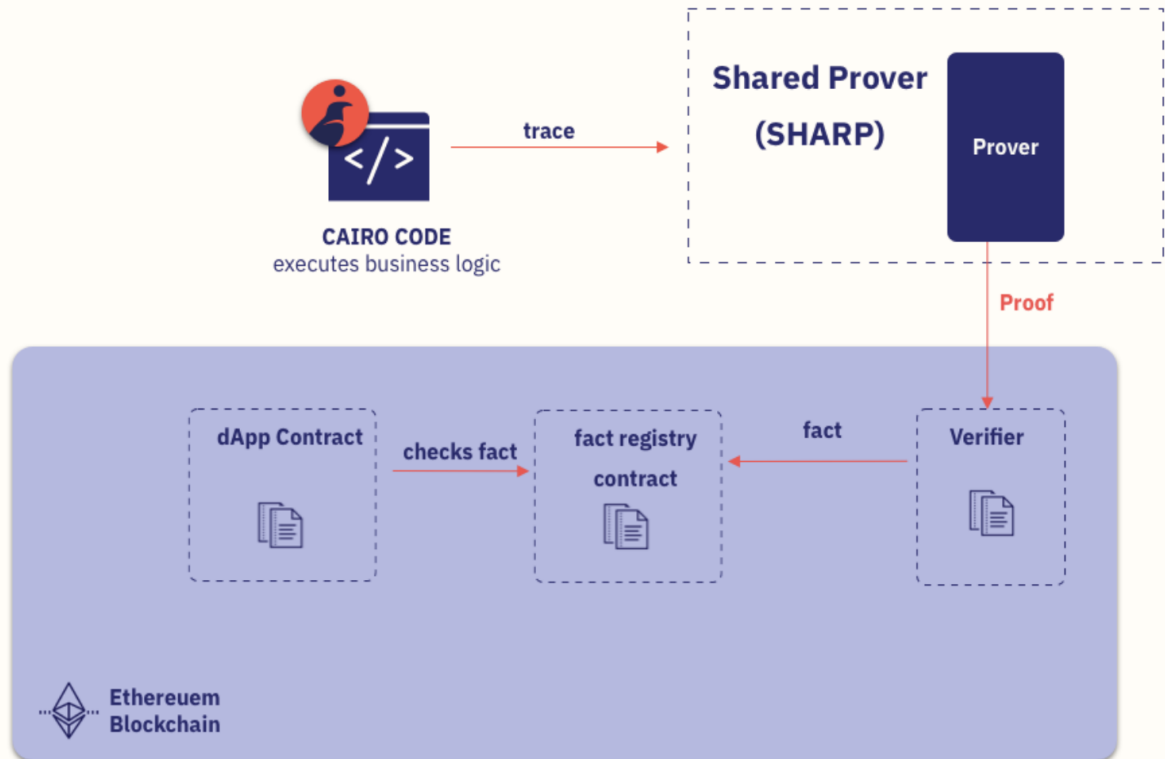
Tuesday : Cairo contracts / Warp

Wednesday : Noir

Thursday : DeFi / ZCash / Aztec

---

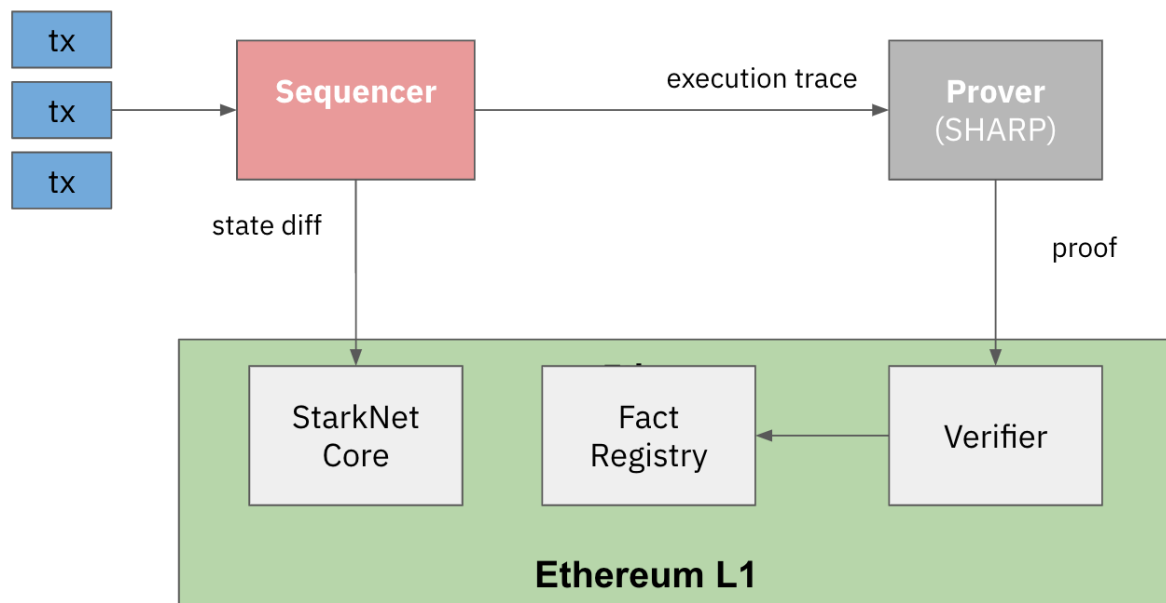
# Shared prover for cairo programs



There are more details of the process [here](#)

# Starknet Architecture

## StarkNet Architecture Overview



See this [article](#) for a good overview

## Starknet Components

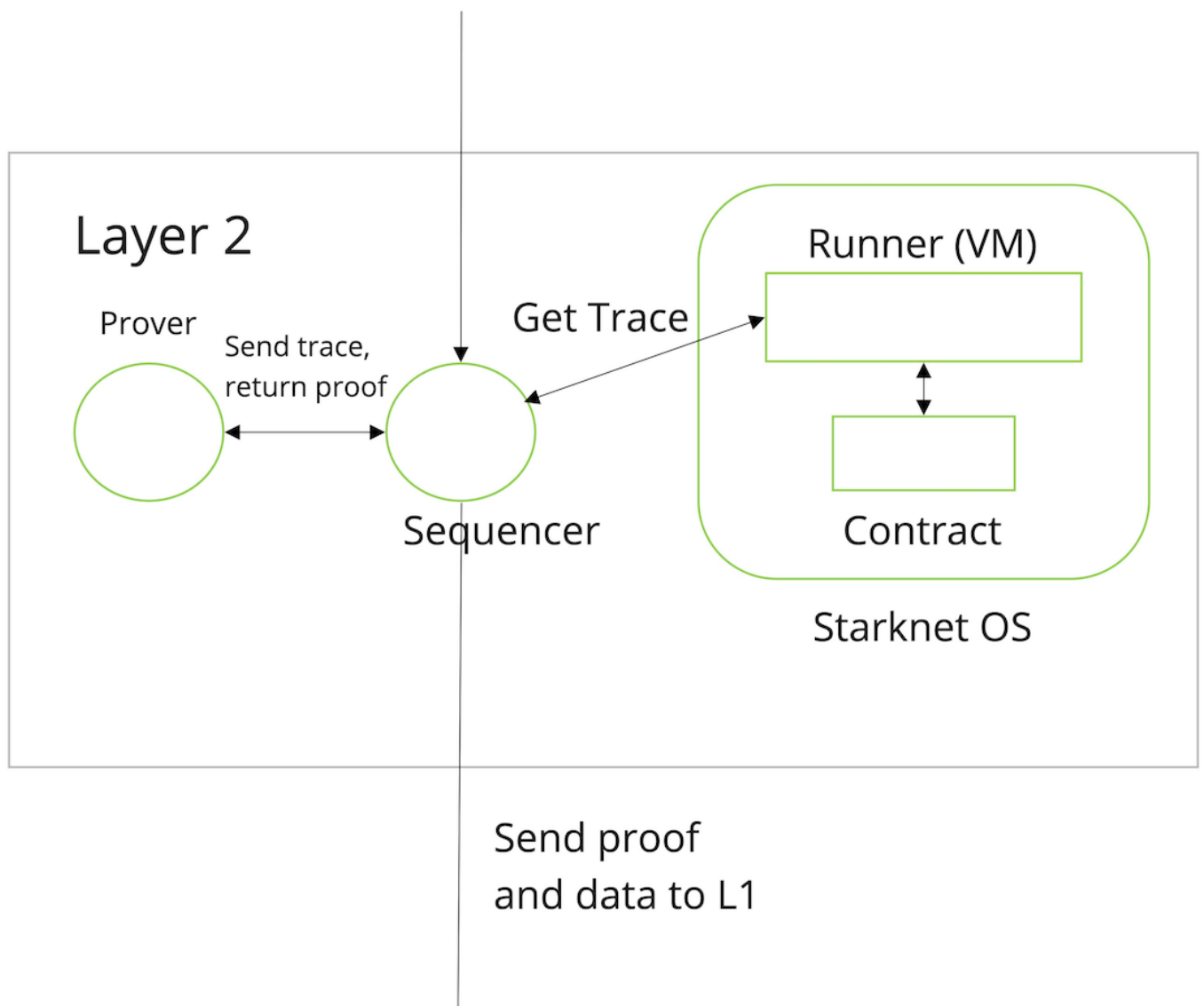
1. **Prover**: A separate process (either an online service or internal to the node) that receives the output of Cairo programs and generates STARK proofs to be verified. The Prover submits the STARK proof to the verifier that registers the fact on L1.
2. **StarkNet OS**: Updates the L2 state of the system based on transactions that are received as inputs. Effectively facilitates the execution of the (Cairo-based) StarkNet contracts. The OS is Cairo-based and is essentially the program whose output is proven and verified using the STARK-proof system. Specific system operations and functionality available for StarkNet contracts are available as calls made to the OS.
3. **StarkNet State**: The state is composed of contracts' code and contracts' storage.
4. **StarkNet L1 Core Contract**: This L1 contract defines the state of the system by storing the commitment to the L2 state. The contract also stores the StarkNet OS program hash – effectively defining the version of StarkNet the network is running.

The committed state on the L1 core contract acts as provides as the consensus mechanism of StarkNet, i.e., the system is secured by the L1 Ethereum consensus. In addition to maintaining the state, the StarkNet L1 Core Contract is the main hub of operations for StarkNet on L1.

Specifically:

- It stores the list of allowed verifiers (contracts) that can verify state update transactions
- It facilitates L1 ↔ L2 interaction

## Transaction



Starknet has blocks, see [Block structure](#) which consists of a header and a set of transactions  
For further details see Starknet [documentation](#)

### Transaction Status

#### 1. NOT\_RECEIVED

Transaction is not yet known to the sequencer

#### 2. RECEIVED

Transaction was received by the sequencer. Transaction will now either execute successfully or be rejected.

#### 3. PENDING

Transaction executed successfully and entered the [pending block](#).

#### 4. REJECTED

Transaction executed unsuccessfully and thus was skipped (applies both to a pending and an actual created block). Possible reasons for transaction rejection:

- An assertion failed during the execution of the transaction (in StarkNet, unlike in Ethereum, transaction executions do not always succeed).
- The block may be rejected on L1, thus changing the transaction status to `REJECTED`

#### 5. `ACCEPTED_ON_L2`

Transaction passed validation and entered an actual created block on L2.

#### 6. `ACCEPTED_ON_L1`

Transaction was accepted on-chain.

Transaction types

1. Deploy
  2. Invoke
  3. Declare
-

# Invoke Transaction Structure

Name	Type	Description
contract_address	FieldElement	The address of the contract invoked by this transaction
entry_point_selector	FieldElement	The encoding of the selector for the function invoked (the entry point in the contract)
calldata	List<FieldElement>	The arguments passed to the invoked function
signature	List<FieldElement>	Additional information given by the caller, representing the signature of the transaction
max_fee	FieldElement	The maximum fee that the sender is willing to pay for the transaction
version	FieldElement	The transaction's version <sup>1</sup>

## Full Nodes

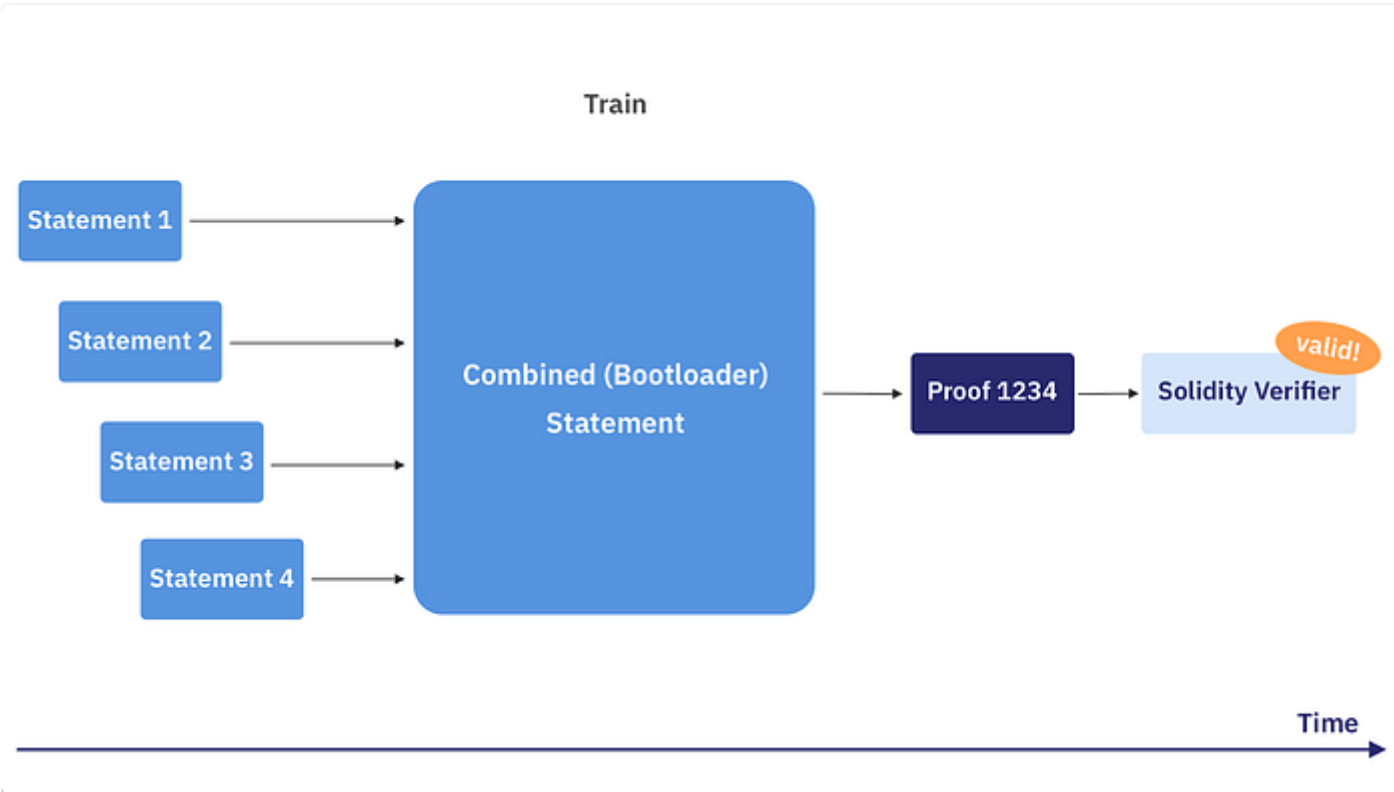
These run the Pathfinder client to keep a record of all the transactions performed in the rollup and to track the current global state of the system.

Full Nodes receive this information through a p2p network where changes in the global state and the validity proofs associated with it are shared everytime a new block is created. When a new Full Node is set up it is able to reconstruct the history of the rollup by connecting to an Ethereum node and processing all the L1 transactions associated with StarkNet.

## Recursive STARKS

See [post](#)

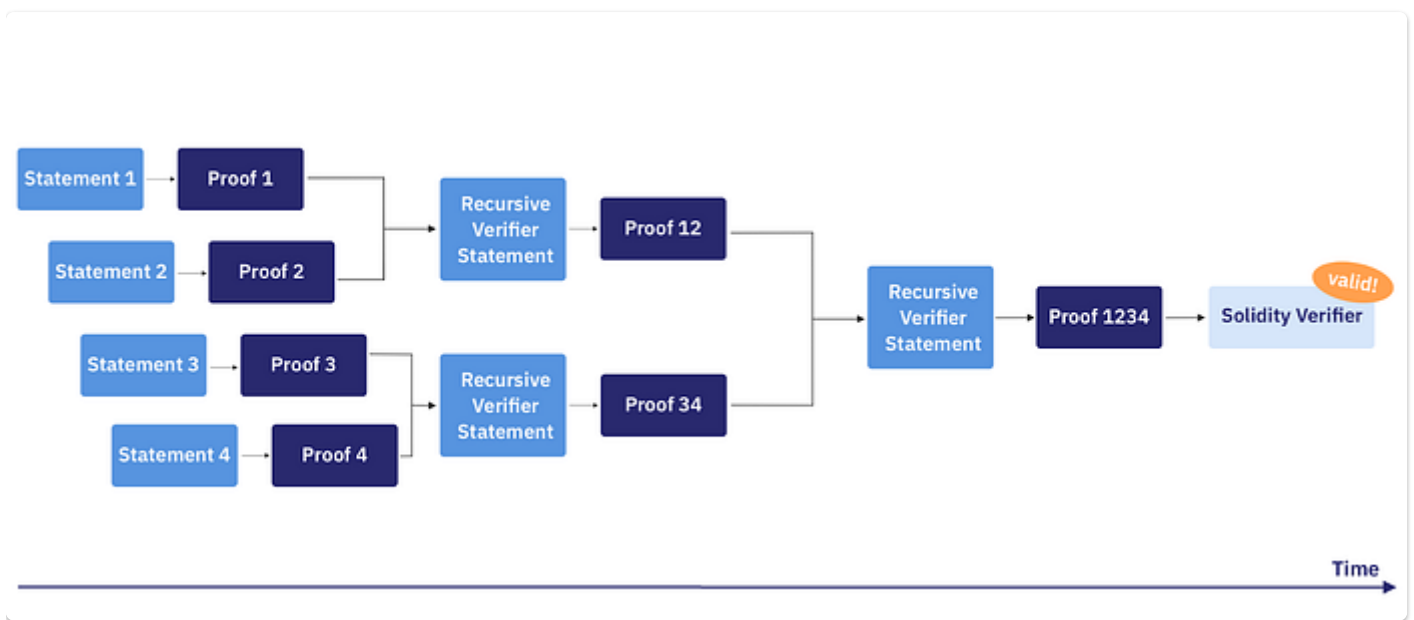
Initially SHARP (The shared prover) would process proofs from applications sequentially, once a threshold number of transactions had arrived, a proof would be generated for all of them.



The amount of memory needed to generate the proof was a limiting factor.

STARKs have roughly linear proving time and log validation time.

## Recursive proofs



Here the proofs are calculated in parallel, then combined in pairs and a proof created and so on.

This results in

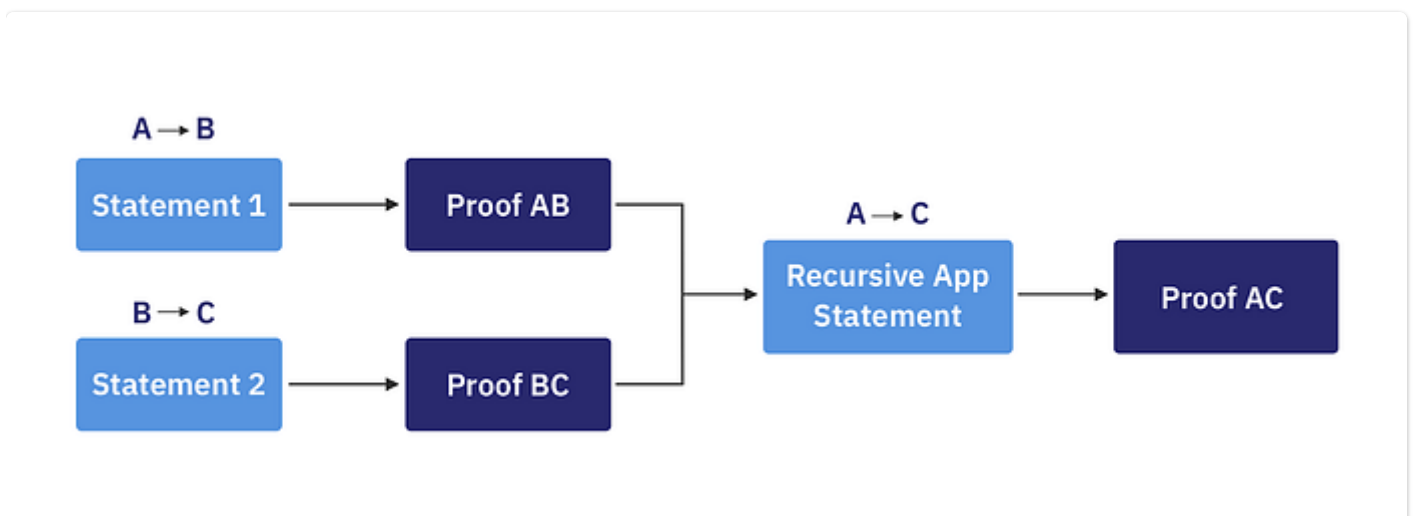
1. Reduced on chain cost, and memory requirements
2. Reduced latency since the proofs can be computed in parallel and we don't need to wait for the final proof to arrive.

## Application recursion

Each STARK proof attests to the validity of a statement applied to some input

STARK recursion compresses two proofs with *two* inputs into *one* proof with two inputs. In other words, while the number of proofs is reduced, the number of inputs is kept constant.

If the recursive statement is allowed to be *application-aware*, i.e. recognizes the semantics of the application itself, it can both compress two proofs into one *as well as* combine the two inputs into one.







# Rust Introduction 2

## Ownership

### Problem

We want to be able to control the lifetime of objects efficiently , but without getting into some unsafe memory state such as having 'dangling pointers'

Rust places restrictions on our code to solve this problem, that gives us control over the lifetime of objects while guaranteeing memory safety.

In Rust when we speak of ownership, we mean that a variable owns a value, for example

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

language-rust

here `my_vec` owns the vector.

(The ownership could be many layers deep, and become a tree structure.)

We want to avoid,

- the vector going out of scope, and `my_vec` ends up pointing to nothing, or (worse) to a different item on the heap
- `my_vec` going out of scope and the vector being left, and not cleaned up.

### Rust ownership rules

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

```
fn main() {  
    {  
        let s = String::from("hello"); // s is valid from this point forward  
  
        // do stuff with s  
    }  
    // this scope is now over, and s is no  
    // longer valid  
}
```

language-rust

### Copying

For simple datatypes that can exist on the stack, when we make an assignment we can just copy the value.

```
fn main() {  
    let a = 2;
```

language-rust

```
let b = a;  
}
```

The types that can be copied in this way are

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain these types. For example, `(i32, i32)`, but not `(i32, String)`.

For more complex datatypes such as `String` we need memory allocated on the heap, and then the ownership rules apply

## Move

For none copy types, assigning a variable or setting a parameter is a `move`. The source gives up ownership to the destination, and then the source is uninitialised.

```
let a=vec![1,2,3];  
let b = a;  
let c = a;    <= PROBLEM HERE
```

Passing arguments to functions transfers ownership to the function parameter

## Control Flow

We need to be careful if the control flow in our code could mean a variable is uninitialised

```
let a = vec![1,2,3];  
while f() {  
    g(a) <= after the first iteration, a has lost ownership  
}  
  
fn g(x : u8)...
```

Fortunately collections give us functions to help us deal with moves

```
let v = vec![1,2,3];  
for mut a in v {  
    v.push(4);  
}
```

# References

References are a flexible means to help us with ownership and variable lifetimes.

They are written

`&a`

If `a` has type `T` , then `&a` has type `&T`

These ampersands represent *references*, and they allow you to refer to some value without taking ownership of it.

Because it does not own it, the value it points to will not be dropped when the reference stops being used.

References have no effect on their referent's lifetime , so a referent will not get dropped as it would if it were owned by the reference.

Using a reference to a value is called 'borrowing' the value.

References must not outlive their referent.

There are 2 types of references

1. A *shared* reference

You can read but not mutate the referent.

You can have as many shared references to a value as you like.

Shared references are clones

2. A *mutable* reference, mean you can read and modify the referent.

If you have a mutable ref to a value, you can't have any other ref to that value active at the same time.

This is following the 'multiple reader or single writer principle'.

Mutable references are denoted by `&mut`

for example this works

```
fn main() {
    let mut s = String::from("hello");

    let s1 = &mut s;
    // let s2 = &mut s;

    s1.push_str(" bootcamp");

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

but this fails

```
fn main() {
    let mut s = String::from("hello");
```

```

    let s1 = &mut s;
    let s2 = &mut s;
    s1.push_str(" bootcamp");

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

```

## Traits

These bear some similarity to interfaces in other languages, they are a way to define the behaviour that a type has and can share with other types, where. behaviour is the methods we can call on that type.

Trait definitions are a way to group method signatures together to define a set of behaviors necessary for a particular purpose.

### Defining a trait

```

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

```

language-rust

### Implementing a trait

```

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}", self.username, self.content)
    }
}

```

language-rust

### Default Implementations

```

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

```

language-rust

### Using traits (polymorphism)

```

pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

```

language-rust

```
}
```

## Packages, crates and modules

A `crate` is a binary or library, a. number of these (plus other resources) can form. a `package`, which will contain a *Cargo.toml* file that describes how to build those crates.

A crate is meant to group together some functionality in a way that can be easily shared with other projects.

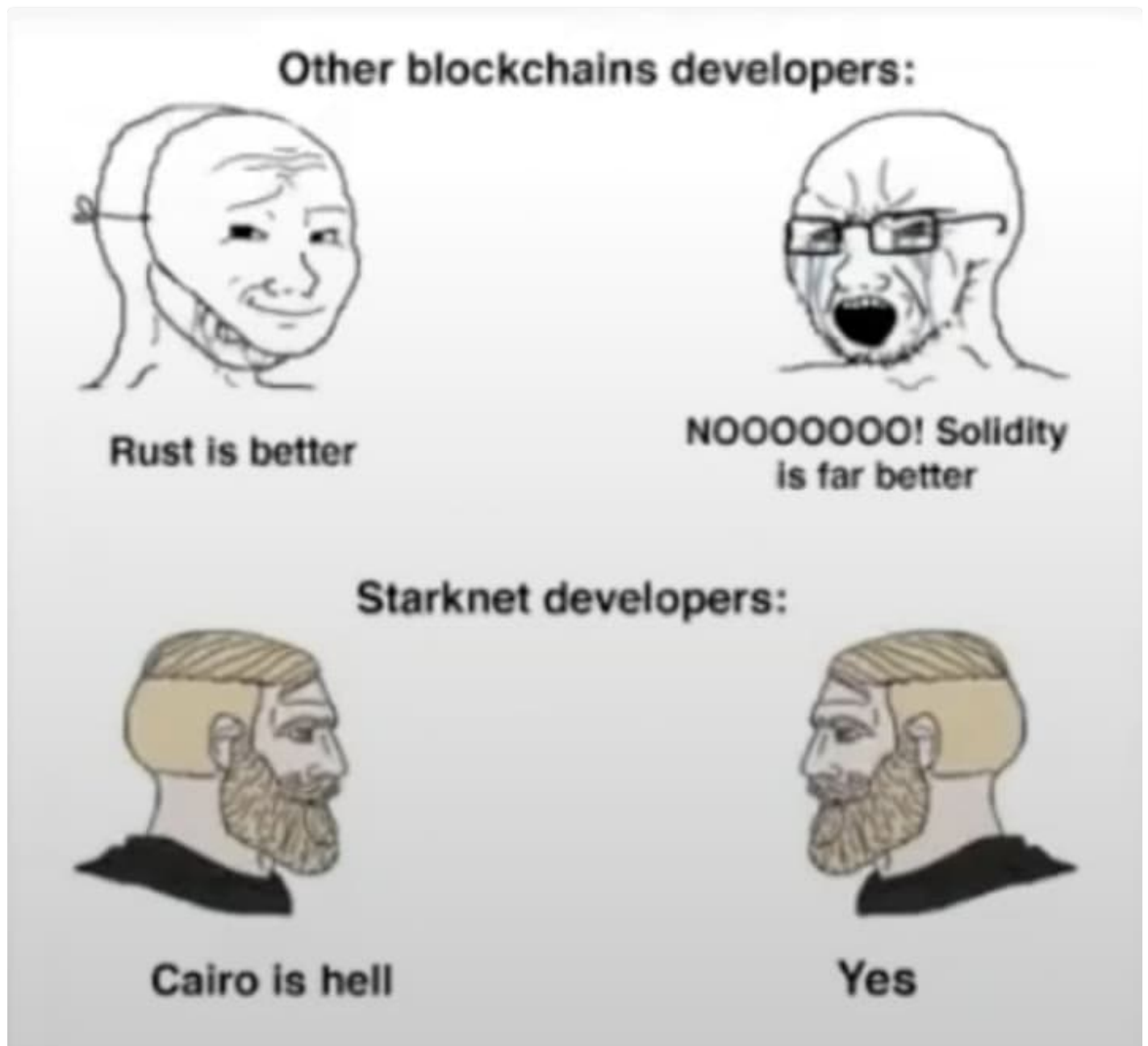
A package can contain at most one library crate, but. any number of binary crates.

There can be further refinement with the use of `modules` which organise code within a crate and can specify the privacy (public or private) of the code.

Module code is private by default, but you can make definitions public by adding the `pub` keyword.

---

# Cairo 1.0



This Cairo 0 meme *should* soon be retired

## Introduction

Cairo is the programming language used for [StarkNet](#). It aims to validate computation and includes the roles of prover and verifier.

It is a Turing complete language.

"In Cairo programs, you write what results are *acceptable*, not *how to* come up with results."

In solidity we might write a statement to extract an amount from a balance, in Cairo we would write a statement to check that for the parties involved the sum of the balances hasn't changed.

## General Points

- Cairo is a Turing complete language for creating STARK-provable programs for general computation.
- It can be approached at a low level, it supports a read-only nondeterministic memory, which means that the value for each memory cell is chosen by the prover, but it cannot change over time.
- There is a distinction between Cairo programs (stateless) and Cairo contracts (given storage in the context of Starknet)
- The Cairo [white paper](#) is more readable than some, but this describes Cairo version 0

## Memory Model

From Cairo [docs](#)

Cairo supports a read-only nondeterministic memory, which means that the value for each memory cell is chosen by the prover, but it cannot change over time (during a Cairo program execution).

## Documentation

The [Cairo book](#) documents Cairo 1.0

# Cairo Language

From the Introduction in the Cairo book

"Cairo is a programming language designed for a virtual CPU of the same name. The unique aspect of this processor is that it was not created for the physical constraints of our world but for cryptographic ones, making it capable of efficiently proving the execution of any program running on it. This means that you can perform time consuming operations on a machine you don't trust, and check the result very quickly on a cheaper machine.

While Cairo 0 used to be directly compiled to CASM, the Cairo CPU assembly, Cairo 1 is a more high level language.

It first compiles to Sierra, an intermediate representation of Cairo which will compile later down to a safe subset of CASM.

The point of Sierra is to ensure your CASM will always be provable, even when the computation fails."

Cairo is based on Rust syntax and semantics, but there are some differences



# Differences between Cairo and Rust

## Loops

Cairo only has one kind of loop for now: `loop`.

```
use debug::PrintTrait;

fn main() {
    let mut counter = 0;

    let result = loop {
        if counter == 10 {
            break counter * 2;
        }
        counter += 1;
    };

    'The result is '.print();
    result.print();
}
```

language-rust

## Collections

```
use dict::DictFeltToTrait; language-rust

fn create_dict() -> DictFeltTo::<felt> {
    let mut d = DictFeltToTrait::new();
    d.insert('First', 1); // Write.
    d.insert('Second', 2);
    d
}

fn main() -> felt {
    let mut dict = create_dict();
    let a = dict.get('First');
    let b = dict.get('Second');
    dict.squash();
    b // returns 2
}
```

`Span` is a struct that represents a snapshot of an `Array`. It is designed to provide safe and controlled access to the elements of an array without modifying the original array. `Span` is particularly useful for ensuring data integrity and avoiding borrowing issues when passing arrays between functions or when performing read-only operations.

All methods provided by `Array` can also be used with `Span`, with the exception of the `append()` method.

### Turning an Array into span

To create a `Span` of an `Array`, call the `span()` method:

```
let span = array.span();
```

## Shadowing

Variable shadowing refers to the declaration of a new variable with the same name as a previous variable.

```
use debug::PrintTrait; language-rust

fn main() {
    let x = 5;
    let x = x + 1;
    {
        let x = x * 2;
        'Inner scope x value is:'.print();
        x.print()
    }
    'Outer scope x value is:'.print();
    x.print();
}
```

# Data types

Cairo is a statically typed so the compiler needs to know the data type.

## Scalar types

- Felts

A field element - `felt252`

Arithmetic on `felt252` is done mod  $p$  with  $p = 2^{251} + 17 * 2^{192} + 1$ . Division doesn't work as you might expect, since we are dealing with integers. In Cairo, the result of  $x/y$  is defined to always satisfy the equation  $(x / y) * y == x$ . If  $y$  divides  $x$  as integers, you will get the expected result in Cairo (for example  $6 / 2$  will indeed result in  $3$ ).

- Integers

Integer types are based on the `felt252` type

type	size
u8	8 bits
u16	16 bits
u32	32 bits
u64	64 bits
u128	128 bits
u256	256 bits
usize	32 bits

`u256` is implemented as a struct

```
#[derive(Copy, Drop, PartialEq, Serde)]                                language-rust

struct u256 {

    low: u128,

    high: u128,

}
```

See [github](#)

- Booleans

Booleans are one `felt252` in size.

## Strings

Strings are not natively supported, but there is a short string literal possible by converting to a `felt252`

```
let short_string = 'Encode ZKP';
```

# Type conversion

You can convert between types with the `try_into` and `into` methods

For example

```
use traits::TryInto;
use traits::Into;
use option::OptionTrait;

fn main(){
    let my_felt = 10;
    let my_u8: u8 = my_felt.try_into().unwrap(); // Since a felt252 might not
    fit in a u8, we need to unwrap the Option<T> type
    let my_u16: u16 = my_felt.try_into().unwrap();
    let my_u32: u32 = my_felt.try_into().unwrap();
    let my_u64: u64 = my_felt.try_into().unwrap();
    let my_u128: u128 = my_felt.try_into().unwrap();
    let my_u256: u256 = my_felt.into(); // As a felt252 is smaller than a
    u256, we can use the into() method
    let my_usize: usize = my_felt.try_into().unwrap();
    let my_felt2: felt252 = my_u8.into();
    let my_felt3: felt252 = my_u16.into();
}
```

language-rust

# Resources

Curated [List](#)

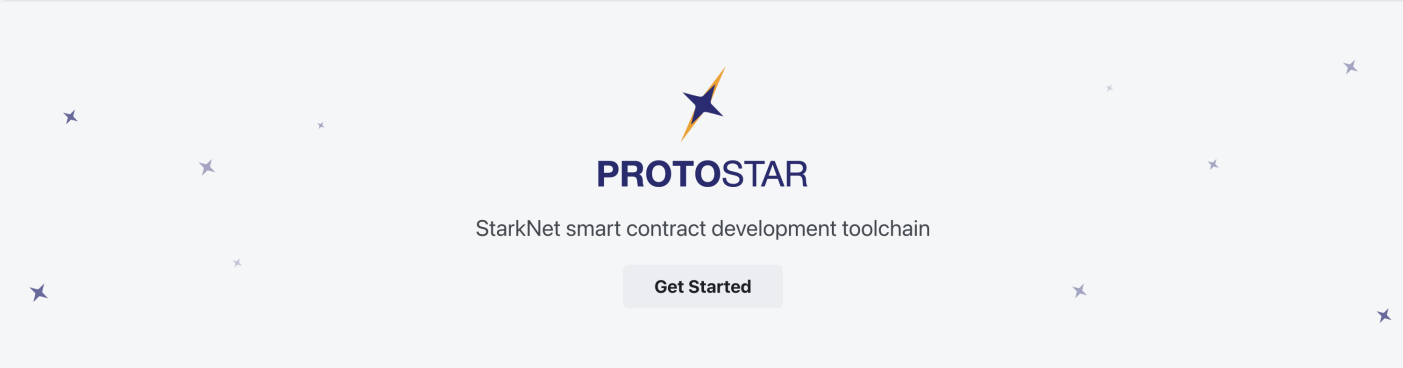
Cairo [Book](#)

You can contribute to Cairo  
see [tweet](#)



# Cairo Development Tools

## Protostar



The banner features the Protostar logo, which consists of a stylized star with a blue and orange gradient. Below the logo, the text "PROTOSTAR" is written in a bold, blue, sans-serif font. Underneath that, "StarkNet smart contract development toolchain" is written in a smaller, grey font. A "Get Started" button is centered below the text. The background is a light grey with several small, faint star icons scattered across it.

**Test**  
Test runner and cheatcodes simplify testing

**Compile**  
No need for setting up Cairo and StarkNet

**Install**  
Add, update, and remove dependencies

See [Protostar](#)

## Features

CLI toolchain  
Unit test programs and contracts  
Deploy contracts

## Plugins

Plugins are available for popular IDEs offering some degree of language support

VSCode [plugin](#) for Cairo :

Hardhat [plugin](#)

[Foundry](#) experimental