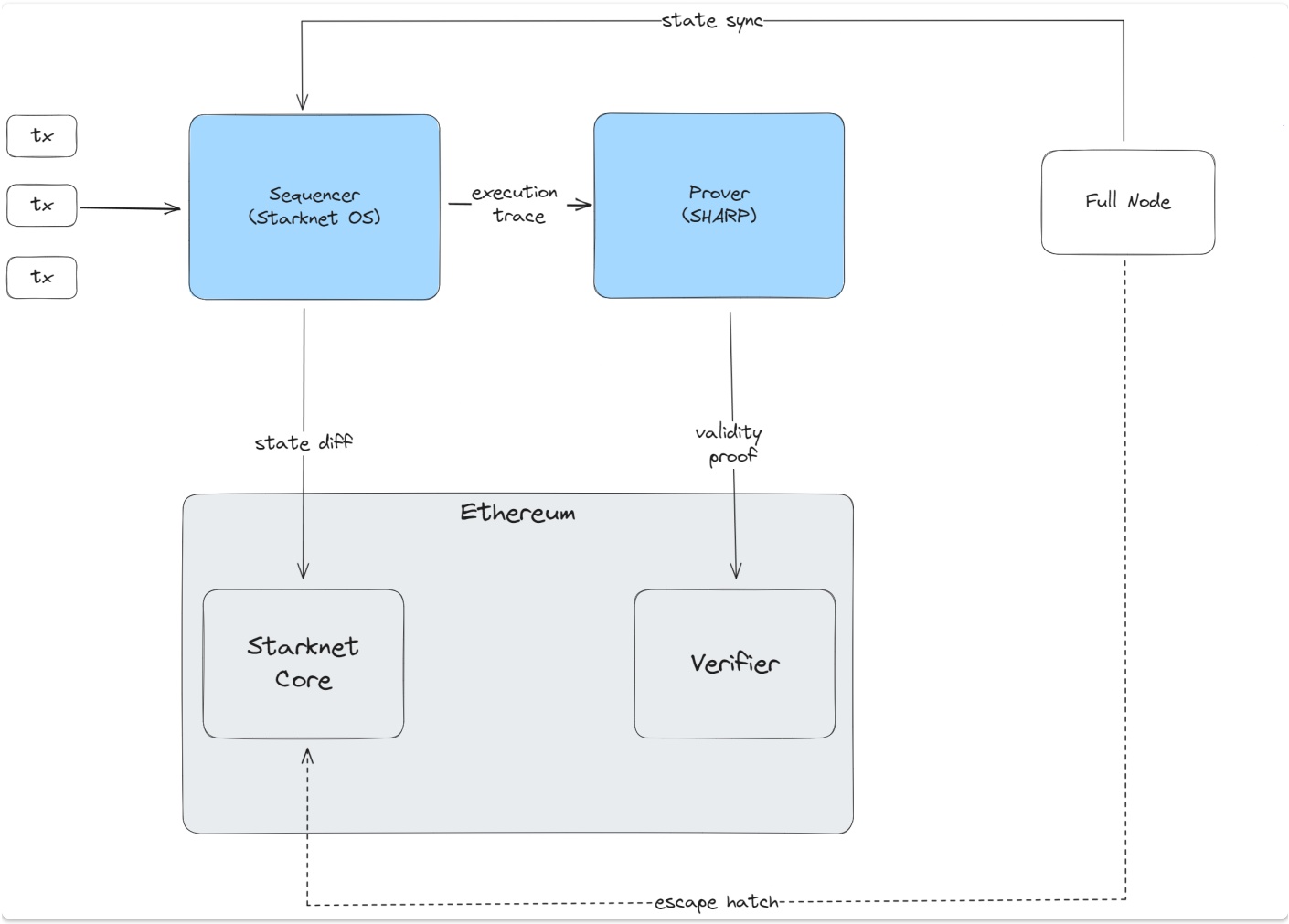
Lesson 5

Starknet Architecture



See this article for a good overview

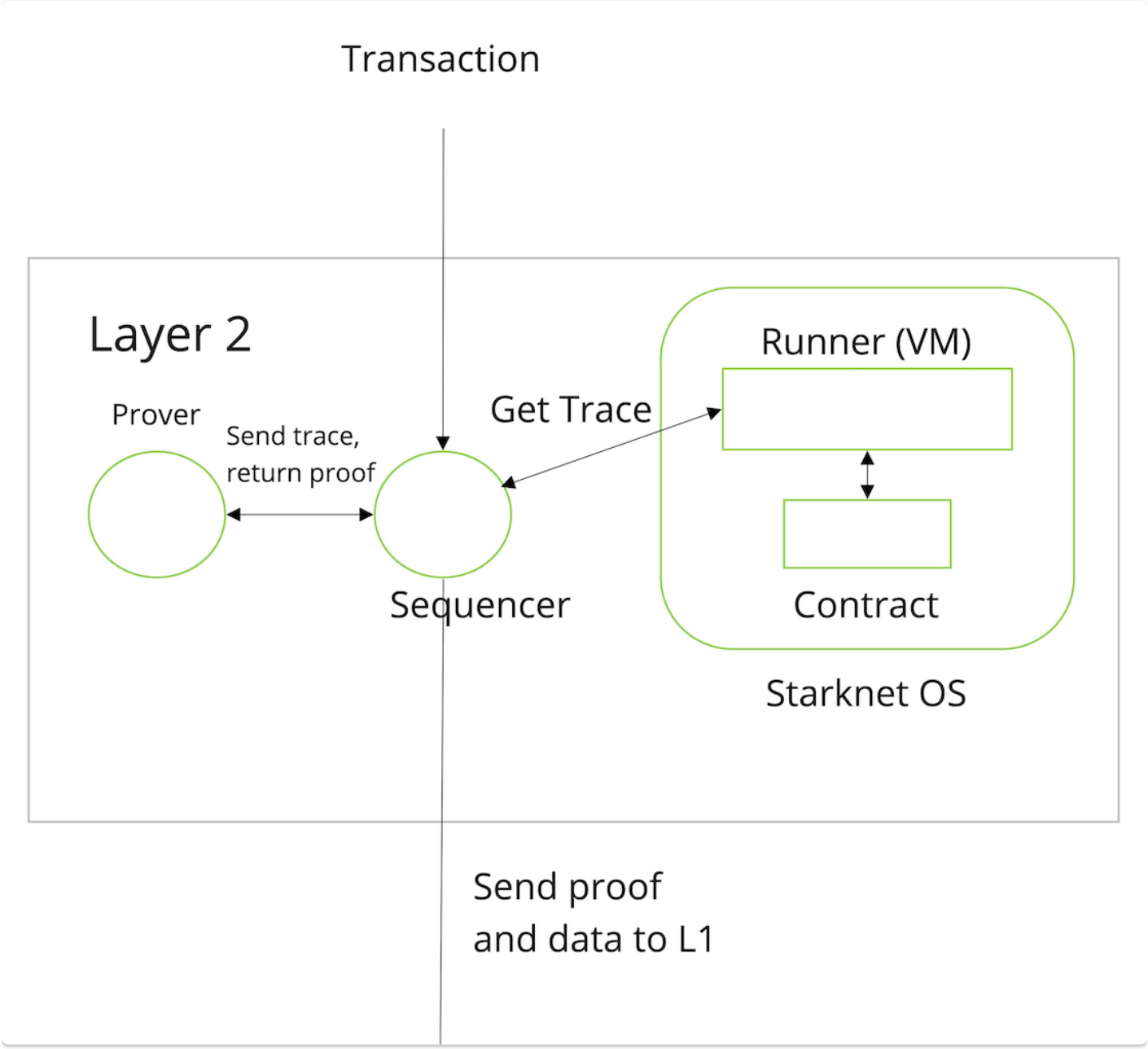
Starknet Components

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

5. **Starknet Full Nodes:** Can get the current state of the network from the sequencer. If the connection between the Sequencer and the Full Node fails for some reason, you can recreate the L2 current state by indexing date from the **Starknet L1 Core Contract** independently



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. ACCEPTED_ON_L2
   - Transaction passed validation and entered an actual created block on L2.
4. ACCEPTED_ON_L1
   - Transaction was accepted on-chain.
5. 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`.

Note that the PENDING in the transaction life-cycle has been removed with the v0.12 Quantum Leap Update. This simplifies the transaction confirmation process and reduce ambiguity. This is because when transactions are submitted to a pending block it has the same finality as a block that was submitted to L2.

Transaction types:

1. invoke transaction
2. declare transaction
3. deploy_account transaction

for more information, see Transaction types

Invoke Transaction Structure

Table 1. Transaction fields

Name	Type	Description
`sender_address`	`FieldElement`	The address of the sender of this transaction.
`calldata`	`List<FieldElement>`	The arguments that are passed to the `validate` and `execute` functions.
`signature`	`List<FieldElement>`	Additional information given by the sender, used to validate the transaction.
`max_fee`	`FieldElement`	The maximum fee that the sender is willing to pay for the transaction
`nonce`	`FieldElement`	The transaction nonce.
`version`	`FieldElement`	The transaction's version. The value is 1. When the fields that comprise a transaction change, either with the addition of a new field or the removal of an existing field, then the transaction version increases.

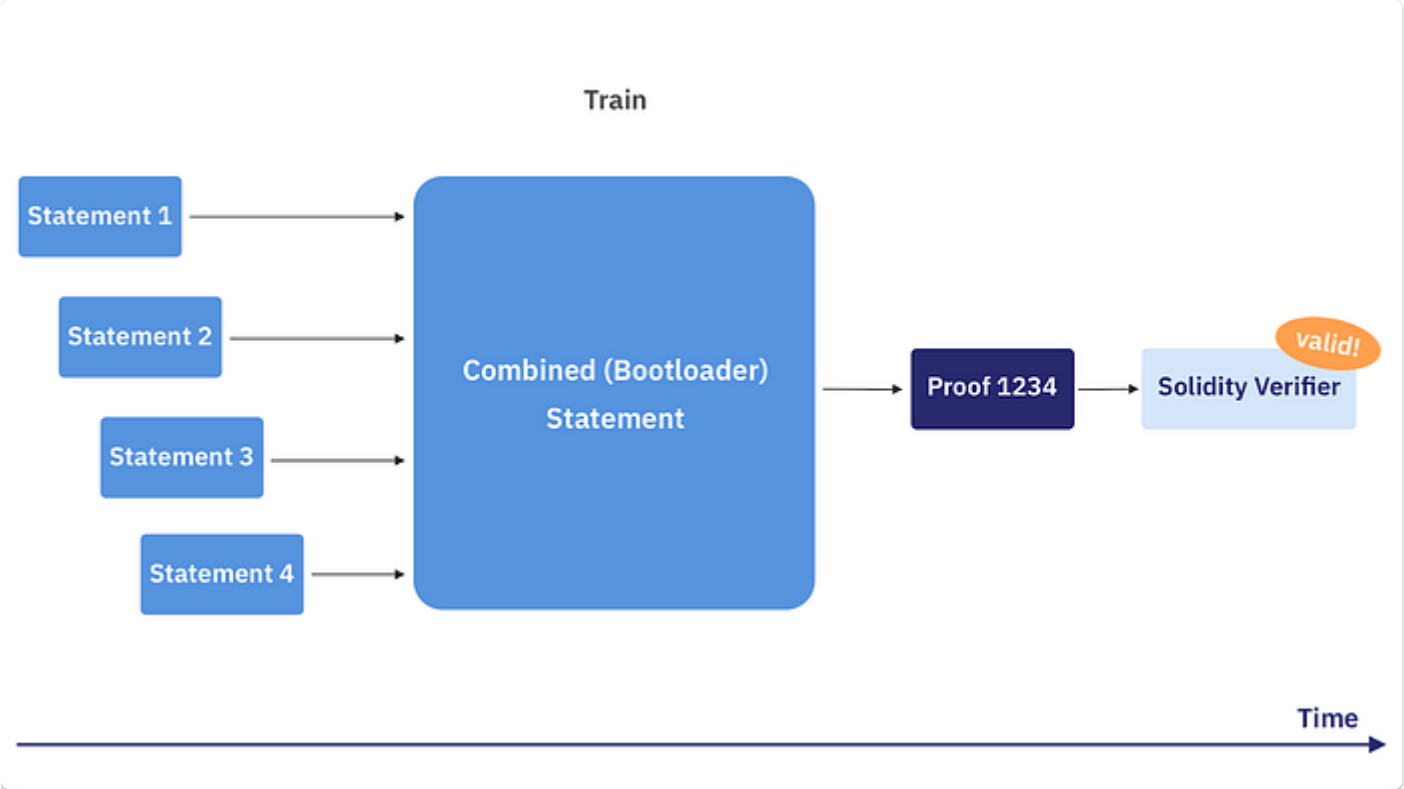
Full Nodes

These run clients such as Pathfinder, Juno or Papyrus to keep a record of all the transactions performed in the Roll 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

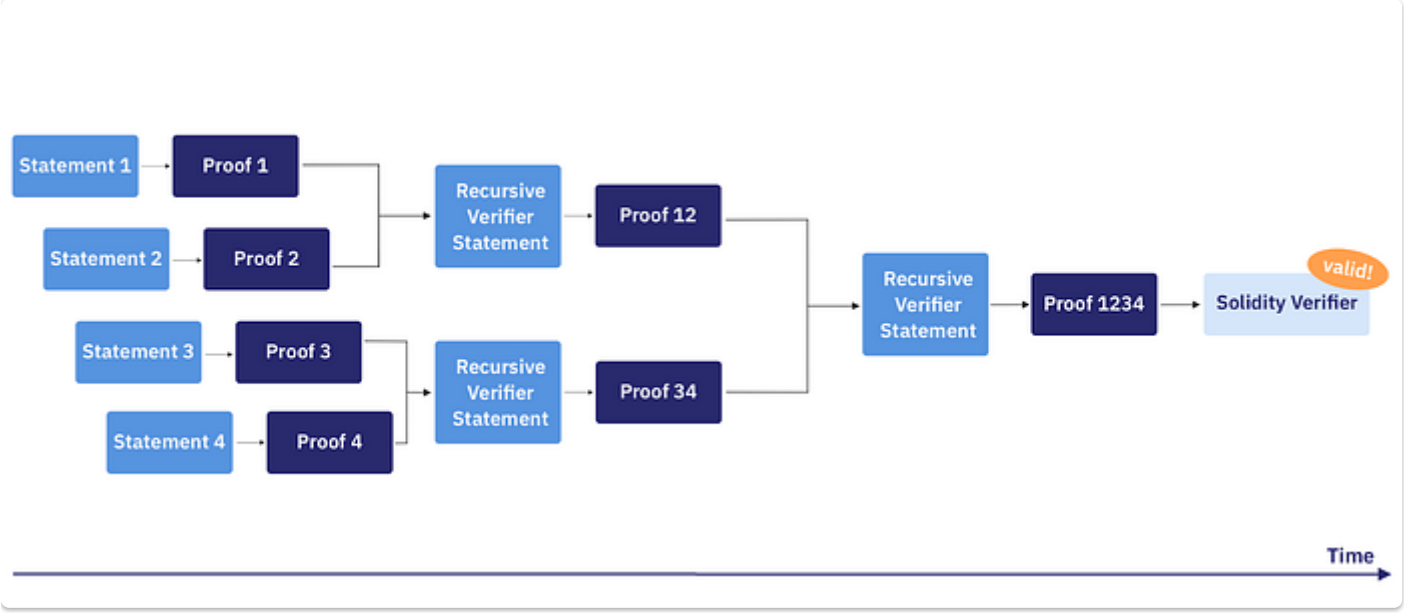
Initially SHARP (The shared prover) would process proofs from applications sequentially, once a threshold number of transactions had arrived, a proof would be generated for all of them.



The amount of memory needed to 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.

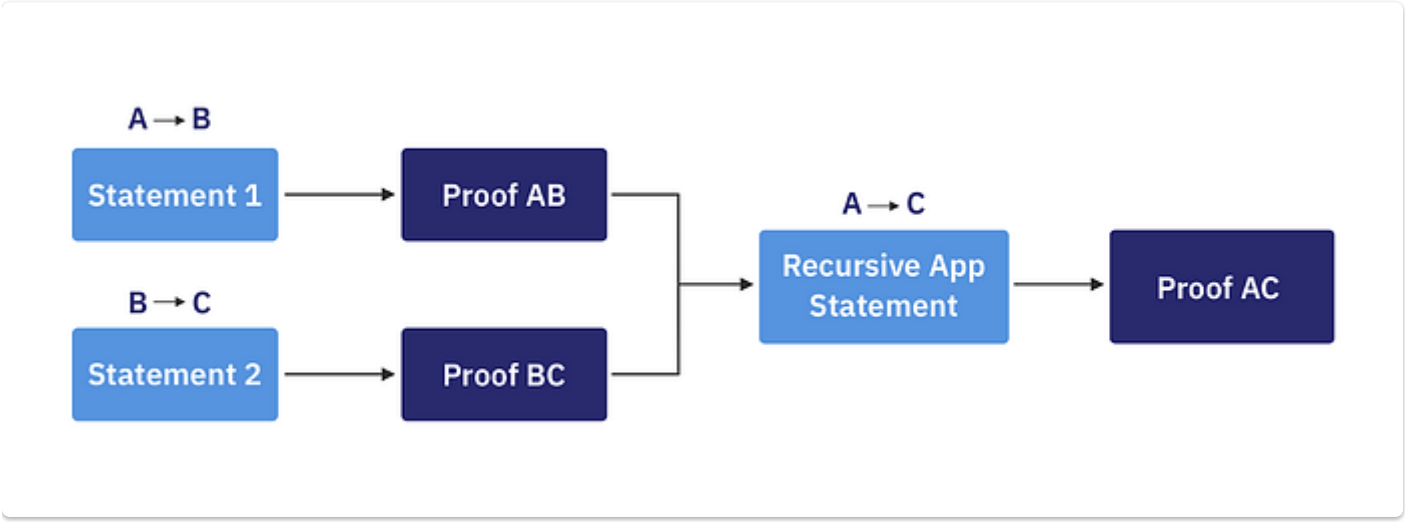
This results in

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 Continued

Idiomatic Rust

The way we design our Rust code and the patterns we will use differ from say what would be used in Python or JavaScript.
As you become more experienced with the language you will be able to follow the patterns

Memory - Heap and Stack

The simplest place to store data is on the stack, all data stored on the stack must have a known, fixed size.
Copying items on the stack is relatively cheap.
For more complex items, such as those that have a variable size, we store them on the heap, typically we want to avoid copying these if possible.

Clearing up memory

The compiler has a simple job keeping track of and removing items from the stack, the heap is much more of a challenge.
Older languages require you to keep track of the memory you have been allocated, and it is your responsibility to return it when it is no longer being used.
This can lead to memory leaks and corruption.
Newer languages use garbage collectors to automate the process of making available areas of memory that are no longer being used. This is done at runtime and can have an impact on performance.

Rust takes a novel approach of enforcing rules at compile time that allow it to know when variables are no longer needed.

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

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

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.

```language-rust
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
}
```

Copying

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

```language-rust
fn main() {
    let a = 2;
    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.

```language-rust
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

```language-rust
let a = vec![1,2,3];
while f() {
        g(a) // <= after the first iteratation, a has lost ownership
}

fn g(x : u8) {
        // ...
}
```

Fortunately collections give us functions to help us deal with moves

```language-rust
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 copies
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

```language-rust
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

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

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

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

De referencing

Use the `*` operator

```rust
let a = 20;
let b = &a;
assert!(*b == 20);
```

String Data types

See Docs

Strings are stored on the heap
A `String` is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to an internal buffer `String` uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

```rust
let len = story.len();
let capacity = story.capacity();
```

We can create a `String` from a literal, and append to it

```rust
let mut title = String::from("Solana ");
title.push_str("Bootcamp"); // push_str() appends a literal to a String
println!("{}", title);
```

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

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

Implementing a trait

```language-rust
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)
    }
}
```

Default Implementations

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

Using traits (polymorphism)

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

Introduction to Generics

Generics are a way to parameterise across datatypes, such as we do below with `Option<T>` where `T` is the parameter that can be replaced by a datatype such as `i32`.
The purpose of generics is to abstract away the datatype, and by doing that avoid duplication.

For example we could have a struct, in this example `T` could be an `i32`, or a `u8`, or ....
depending how you create the Point `struct` in the main function.
In this case we are enforcing `x` and `y` to be of the same type.

```language-rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let my_point = Point { x: 5, y: 6 };
}
```

The handling of generics is done at compile time, so there is no run time cost for including generics in your code.

Introduction to Vectors

Vectors are one of the most used types of collections.

Creating the Vector

We can use `Vec::new()` To create a new empty vector

```language-rust
let v: Vec<i32> = Vec::new();
```

We can also use a macro to create a vector from literals, in which case the compiler can determine the type.

```language-rust
let v = vec![41, 42, 7];
```

Adding to the vector

We use `push` to add to the vector, for example

`v.push(19);`

Retrieving items from the vector

2 ways to get say the 5th item

- using `get`
  e.g. `v.get(4);`
- using an index
  e.g. `v[4];`

We can also iterate over a vector

```language-rust
    let v = vec![41, 42, 7];
     for ii in &v {
         println!("{}", ii);
     }
```

You can get an iterator over the vector with the `iter` method

```language-rust
let x = &[41, 42, 7];
let mut iterator = x.iter();
```

There are also methods to `insert` and `remove`
For further details see Docs

Iterators

The iterator in Rust is optimised in that it has no effect until it is needed

```language-rust
let names = vec!["Bob", "Frank", "Ferris"];
let names_iter = names.iter();
```

This creates an iterator for us that we can then use to iterate through the collection using `.next()`

```language-rust
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];
        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

Shadowing

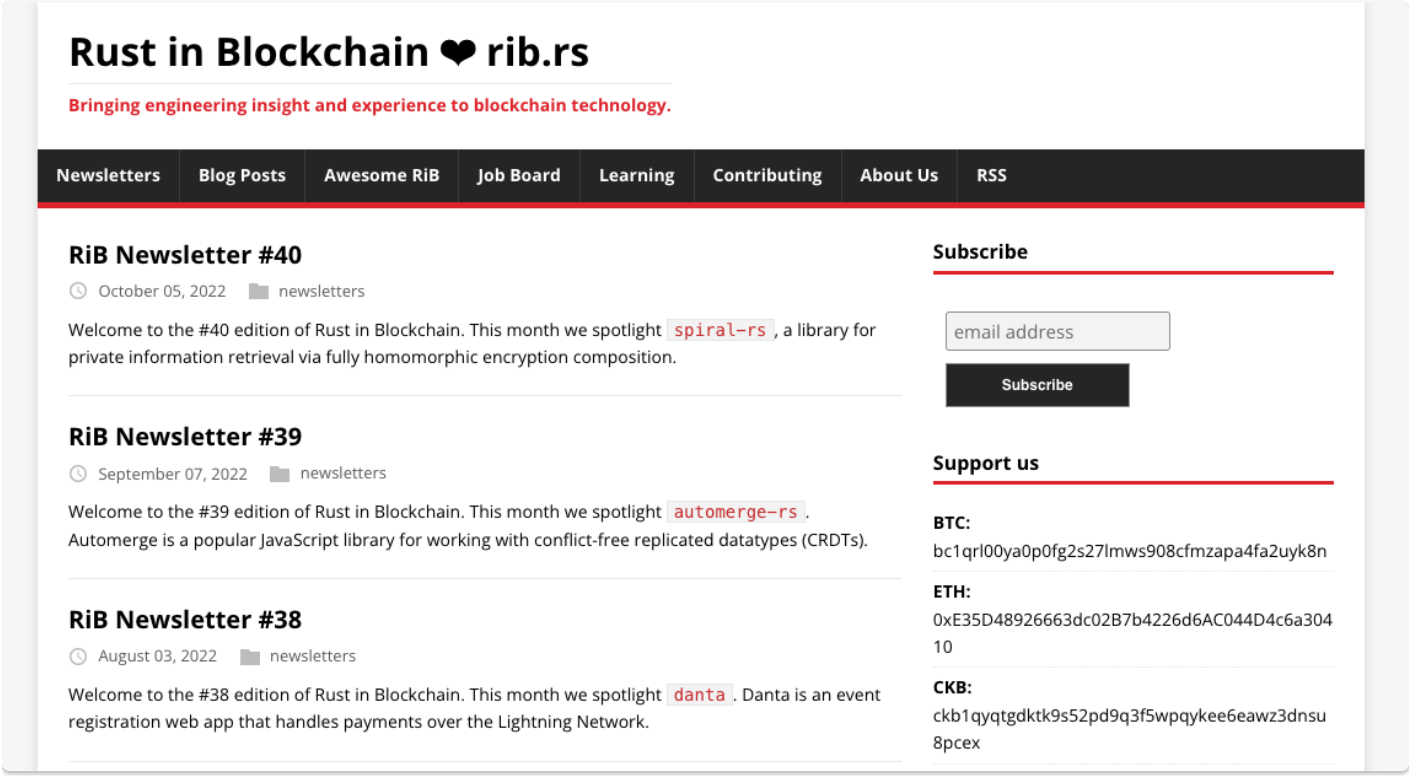
It is possible to declare a new variable with the same name as a previous variable.
The first variable is said to be shadowed by the second,
For example

```rust
fn main() {
    let y = 2;
    let y = y * 3;

    {
        let y = y * 2;
        println!("Here y is: {y}");
    }

    println!("Here y is: {y}");
}
```

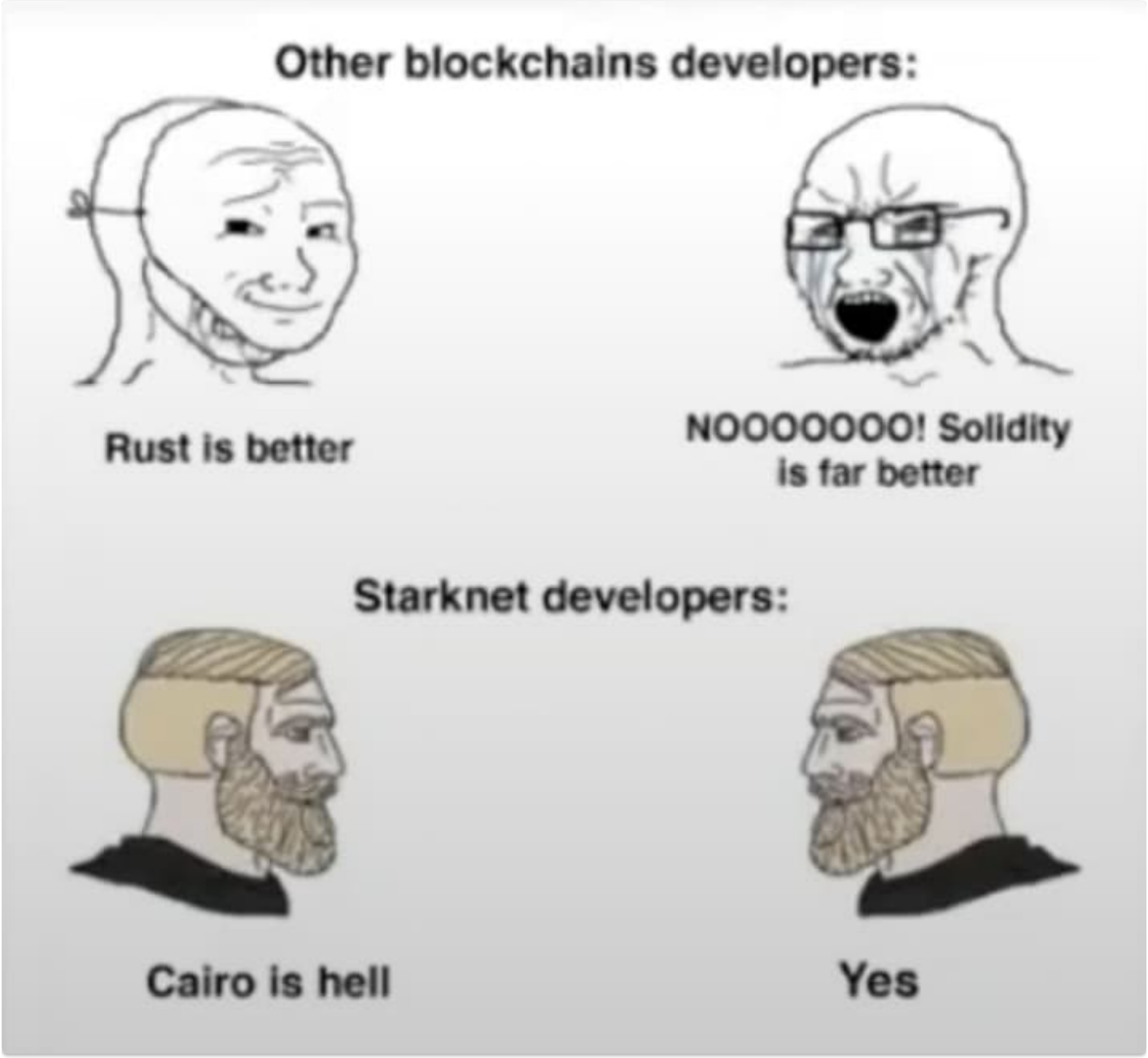
We can use this approach to change the value of otherwise immutable variables

Resources

- Rust [cheat sheet]
- [Rustlings]
- Rust by [example]
- Rust Lang [Docs]
- Rust [Playground]
- Rust [Forum]
- Rust [Discord]
- [Rust in Blockchain]



Cairo



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 the Cairo language

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`.

```language-rust
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();
}
```

Collections

```language-rust
use dict::Felt252DictTrait;

fn main() {
    let mut balances: Felt252Dict<u64> = Default::default();

    balances.insert('Alex', 100);
    balances.insert('Maria', 200);

    let alex_balance = balances.get('Alex');
    assert(alex_balance == 100, 'Balance is not 100');

    let maria_balance = balances.get('Maria');
    assert(maria_balance == 200, 'Balance is not 200');
}
```

`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

See Docs

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

```language-rust
let span = array.span();
```

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

```language-rust
#[derive(Copy, Drop, PartialEq, Serde)]
struct u256 {
    low: u128,
    high: u128,
}
```

See github

- Booleans
  Booleans are one `felt252` in size.

Strings

Strings are not natively supported yet, but there is a short string literal possible by converting to a `felt252`. The maximum length of a short string is 31 characters.

```language-rust
let short_string = 'Encode ZKP';
```

Type conversion

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

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

fn main() {
    let my_felt252 = 10;
    // Since a felt252 might not fit in a u8, we need to unwrap the Option<T>
type
    let my_u8: u8 = my_felt252.try_into().unwrap();
    let my_u16: u16 = my_u8.into();
    let my_u32: u32 = my_u16.into();
    let my_u64: u64 = my_u32.into();
    let my_u128: u128 = my_u64.into();
    // As a felt252 is smaller than a u256, we can use the into() method
    let my_u256: u256 = my_felt252.into();
    let my_usize: usize = my_felt252.try_into().unwrap();
    let my_other_felt252: felt252 = my_u8.into();
    let my_third_felt252: felt252 = my_u16.into();
}
```

Resources

Curated List
Cairo Book

Cairo-by-Example - https://cairo-by-example.com/
Starknet-by-Example - https://starknet-by-example.voyager.online/
Node Guardians Quest - https://nodeguardians.io/dev-hub?s=devhub-campaigns&sc=starting-cairo

You can contribute to Cairo
see tweet

Cairo Installation

Scarb installation

Follow the installation steps from here

It is recommended to install it via `asdf` as this will help us switch seamlessly to a upgrade/downgrade `scarb` whenever we need.

For now, please install `scarb v0.5.1` in order to declare and deploy contracts on Starknet.

```
  asdf plugin add scarb https://github.com/software-mansion/asdf-scarb.git
asdf install scarb 0.5.1
asdf global scarb 0.5.1
asdf reshim scarb
```

Once you have completed the installation, we can check if it was successful. Run the following command:

```language-bash
scarb --version
>>
scarb 0.5.1 (798acce7f 2023-07-05)
cairo: 2.0.1 (https://crates.io/crates/cairo-lang-compiler/2.0.1)
```

You should receive the version of Scarb along with the version of Cairo.

Language Server Installation

Next, to ensure everything works with Scarb, we just need to either visit the VS Code marketplace or search for `Cairo 1.0` in your VS Code extensions tab and install it.

Once the extension is installed, Scarb should automatically detect it, and the language server should start functioning properly.

Starkli CLI (optional)

Starkli CLI is used to declare and deploy your smart contracts on Starknet. For more information about the installation process please follow the steps from here.

Once installed, run the following command to verify if the installation was successful:

```language-bash
starkli --version
>>
starkli 0.1.8 (d9bb4f7)
```

If the installation has been successful, you should see the installed version of Starkli displayed.

In case you want to find out more about Starkli, you can check the Starknet Community Call #48