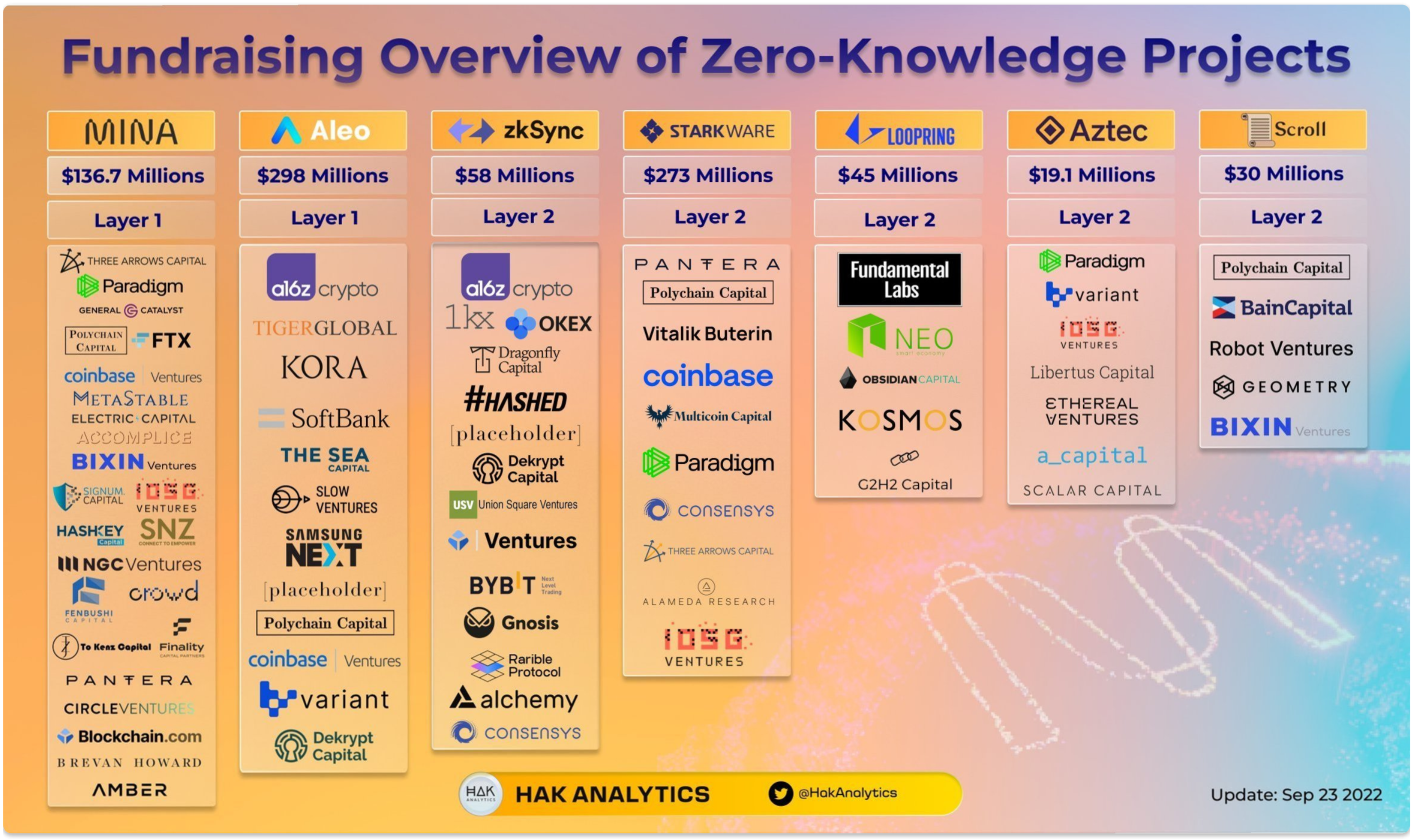
Lesson 4

Cairo / Starknet Introduction

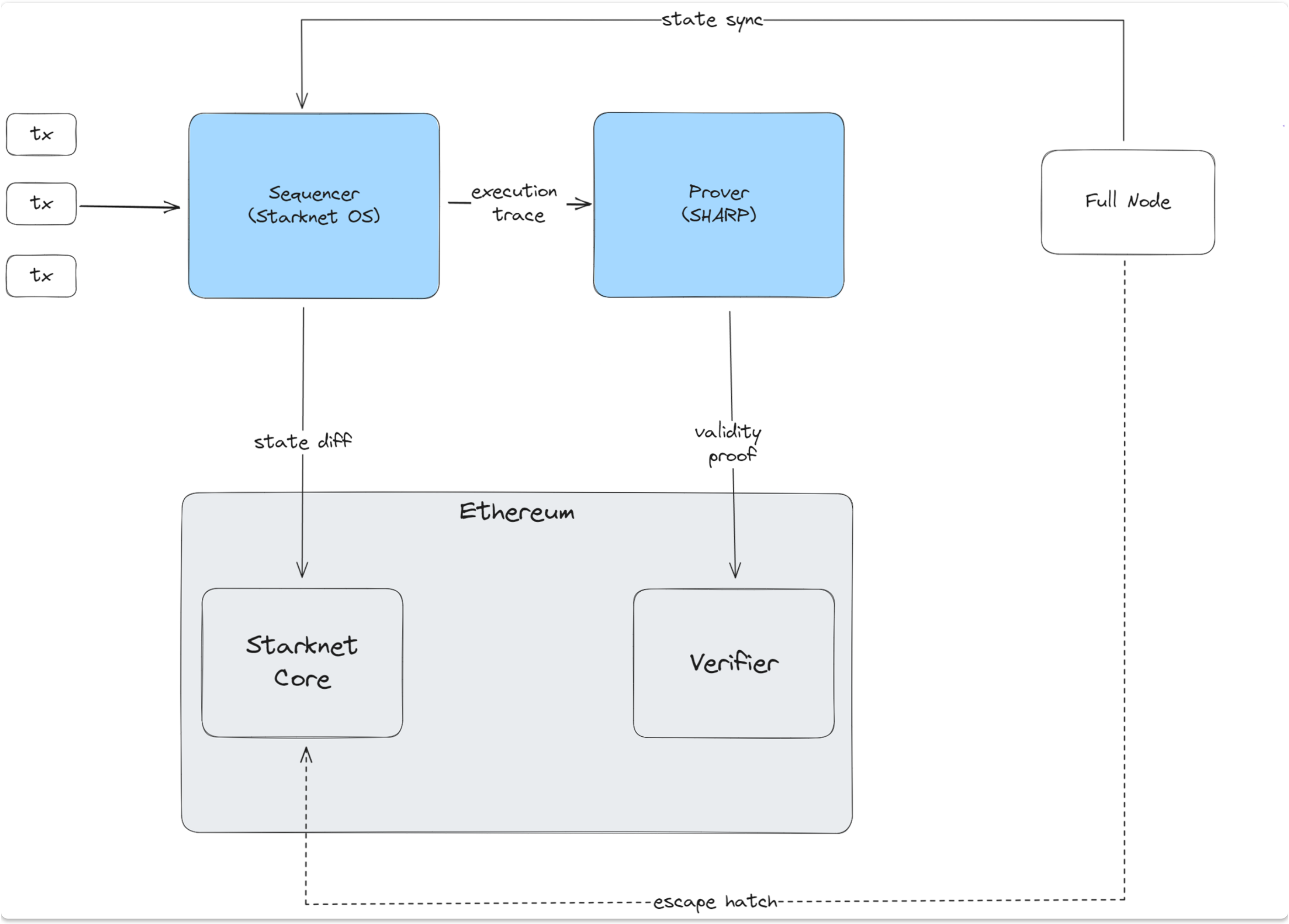


Starknet

"Starknet is a permissionless decentralised ZK-Rollup. It operates as an L2 network over Ethereum, enabling any dApp to achieve unlimited scale for its computation – without compromising Ethereum's composability and security."

See ecosystem

Starknet Architecture Overview



Starknet Components

1. Prover: A separate process (either an online service or internal to the node) that receives the execution trace from the Sequencer 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

---

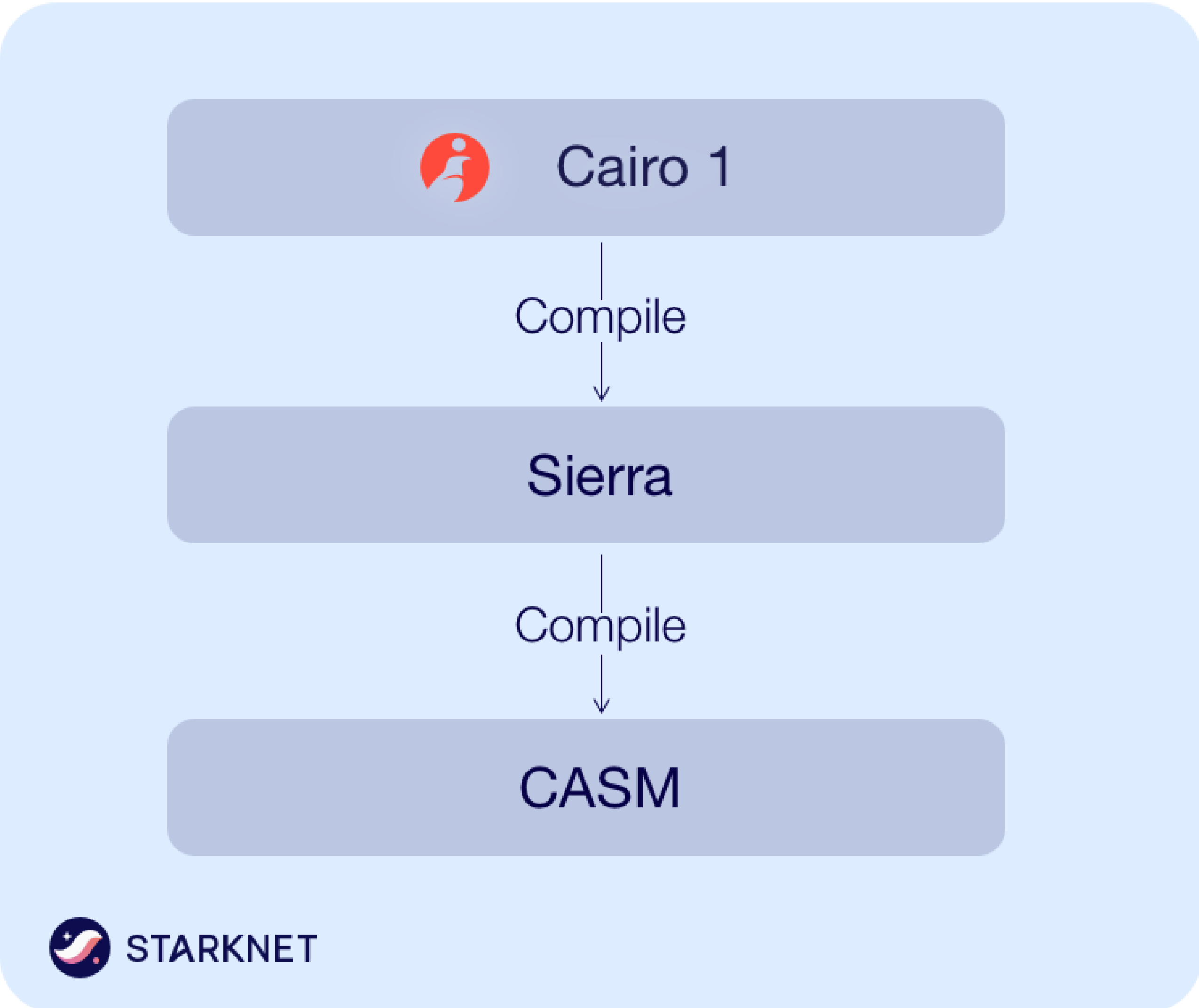
Safe Intermediate Representation (Sierra)

A new intermediate level representation
Transactions should always be provable, even when a transaction fails
Asserts are converted to if statements, if it returns false we don't do any modifications to storage

Contracts will count gas
Still needs to be low level enough to be efficient
So the process would be
Cairo Smart Contract => Sierra => Cairo Assembly => Validity Proof

Sierra bytecode

- cannot fail
- counts gas
- compiles to Cairo with virtually no overhead



Starknet's current roadmap:

[Starknet Roadmap]

Building now

Building now (2)

Builtins and Dynamic Layouts

Q3 2023

0.12.1 | Increased prover efficiency

Inclusion of failed transactions in a block

Q3 2023

v0.12.1 | UX & transaction reliability

Upcoming

Building next (2)

Full node P2P communication & increased throughput

Q3 2023

v0.12.0.2 | Full node P2P communication

Reduction in transaction costs via Volition mode

Q4 2023

v0.13.0 | Reduced transaction costs

Backlog

Backlog (3)

Fee market for transactions
Q4 2023
v0.14.0 · Faster finality

Prove failed transactions
Q4 2023
v0.15.0 · Improved protocol soundness

Short & fixed block intervals
Q4 2023
v0.15.0 · Faster finality

Cairo Versions

We are entering a transition period, and will be teaching Cairo 2.0

Environment	Starknet version	Cairo version
Mainnet	0.12.0	2.0.0
Goerli Testnet 1	0.12.0	2.0.0
Goerli Testnet 2	0.12.0	2.0.0

As of July 12 2023 Starknet v0.12.0 is live on mainnet
See Release notes

Starknet v0.12 comes with a lot of new changes.

- Use the rust blockifier and LambdaClass's Cairo VM to accelerate the sequencer's time to handle transactions.
- Support version 2.0.0 of the Cairo compiler.

- Replace the PENDING status of transactions to ACCEPTED_ON_L2 - once a transaction is in that status it means that it will be included in a block, this applies to transactions - blocks still have the PENDING status.
- Add an experimental get_block_hash syscall.
- Change HTTP error code from 500 to 400 on API error

Some articles about version 2.0

Extropy [Starknet v0.12 Quantum Leap]

Starkware - [Quantum Leap]

> Starknet Forum - [Syntax Proposal]

Introduction to Rust

Core Features

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead

Variables

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier

```rust
let x = 1;
let mut y = 1;
```
language-rust

Types

[Data Types -Rust book]()

The Rust compiler can infer the types that you are using, given the information you already gave it.

[Scalar Types]()

A *scalar* type represents a single value. Rust has four primary scalar types:

- integers
- floating-point numbers
- booleans
- characters

Integers

For example

```rust
u8, i32, u64
```
language-rust

Floating point

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively.
The default type is `f64` because on modern CPUs it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

boolean

The boolean type or bool is a primitive data type that can take on one of two values, called true and false . (size of 1 byte)

char

`char` in Rust is a unique integral value representing a Unicode Scalar value
Note that unlike C, C++ this cannot be treated as a numeric type.

Other scalar types

usize

`usize` is pointer-sized, thus its actual size depends on the architecture your are compiling your program for
As an example, on a 32 bit x86 computer, `usize = u32`, while on x86_64 computers, `usize = u64`.
`usize` gives you the guarantee to be always big enough to hold any pointer or any offset in a data structure, while `u32` can be too small on some architectures.
Rust states the size of a type is not stable in cross compilations except for primitive types.

Compound Types

Compound types can group multiple values into one type.

- tuples
- arrays
- struct

Tuples

Example

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Struct

```rust
struct User {
        name : String,
        age: u32,
        email: String,
}
```

Collections

- Vectors

```rust
let names = vec!["Bob", "Frank", "Ferris"];
```

We will cover these in more detail later

Strings

Based on UTF-8 - Unicode Transformation Format

Two string types:

- `&str` a view of a sequence of UTF8 encoded dynamic bytes, stored in binary, stack or heap. Size is unknown and it points to the first byte of the string
- `String`: growable, mutable, owned, UTF-8 encoded string. Always allocated on the heap. Includes capacity i.e. memory allocated for this string.

A String literal is a string slice stored in the application binary (i.e. there at compile time).

String vs str

`String` - heap allocated, growable UTF-8
`&str` - reference to UTF-8 string slice (could be heap, stack ...)

String vs &str - StackOverflow
Rust overview - presentation
Let's Get Rusty - Strings

Arrays

Rust book definition of an array:

"An array is a collection of objects of the same type `T`, stored in contiguous memory. Arrays are created using brackets `[]`, and their length, which is known at compile time, is part of their type signature `[T; length]`."

Array features:

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Array declarations

```
//Syntax1: No type definition
let variable_name = [value1,value2,value3];

let arr = [1,2,3,4,5];

//Syntax2: Data type and size specified
let variable_name:[dataType;size] = [value1,value2,value3];

let arr:[i32;5] = [1,2,3,4,5];

//Syntax3: Default valued array
let variable_name:[dataType;size] =
[default_value_for_elements,size];

let arr:[i32;3] = [0;3];

// Mutable array
let mut arr_mut:[i32;5] = [1,2,3,4,5];
```

```rust
// Immutable array
let arr_immut:[i32;5] = [1,2,3,4,5];
```

Rust book definition of a slice:

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.

Arrays - TutorialsPoint
Arrays and Slices - RustBook

Numeric Literals

The compiler can usually infer the type of an integer literal, but you can add a suffix to specify it, e.g.
`42u8`
It usually defaults to `i32` if there is a choice of the type.

Hexadecimal, octal and binary literals are denoted by prefixes
`0x` , `0o`, and `0b` respectively

To make your code more readable you can use underscores with numeric literals e.g.

```rust
1_234_567_890
```

ASCII code literals

Byte literals can be used to specify ASCII codes
e.g.
`b'C'`

Conversion between types

Rust is unlike many languages in that it rarely performs implicit conversion between numeric types, if you need to do that, it has to be done explicitly.
To perform casts between types you use the `as` keyword
For example

```rust
let a = 12;
let b = a as usize;
```

Enums

See docs

Use the keyword `enum`

```rust
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```rust
Fruit::Orange
```

Functions

Functions are declared with the `fn` keyword, and follow familiar syntax for the parameters and function body.

```rust
fn my_func(a: u32) -> bool {
    if a == 0 {
        return false;
    }
    a == 7
}
```

As you can see the final line in the function acts as a return from the function Typically the return keyword is used where we are leaving the function before the end.

Loops

Range:

- inclusive start, exclusive end
  `for n in 1..101 {}`
- inclusive end, inclusive end
  `for n in 1..=101 {}`
- - inclusive end, inclusive end, every 2nd value

```rust
for n in (1..=101).step_by(2){}
```

We have already seen for loops to loop over a range, other ways to loop include
`loop` - to loop until we hit a `break`
`while` which allows an ending condition to be specified
See [Rust book] for examples.

Control Flow

If expressions

See Docs

The `if` keyword is followed by a condition, which *must evaluate to bool* , note that Rust does not automatically convert numerics to bool.

```rust
if x < 4 {
        println!("lower");
} else {
        println!("higher");
}
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Printing

```rust
println!("Hello, world!");

println!("{:?} tokens", 19);
```

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library.
The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

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

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```rust
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

The keyword `match` is followed by an expression, in this case `coin`
The value of this is matched against the 'arms' in the expression.
Each `arm` is made of a pattern and some code
If the value matches the pattern, then the code is executed, each arm is an expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```rust
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

Installing Rust

The easiest way is via rustup
See Docs

Mac / Linux

```bash
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

language-bash

Windows
See details here
download and run `rustup-init.exe`.

Other methods

Cargo

See the docs

Cargo is the rust package manager, it will

- download and manage your dependencies,
- compile and build your code
- make distributable packages and upload them to public registries.

Some common cargo commands are (see all commands with --list):

```
  build, b      Compile the current package
check, c      Analyse the current package and report errors, but
don't build
                          object files
clean         Remove the target directory
doc, d        Build this package's and its dependencies'
documentation
new           Create a new cargo package
init          Create a new cargo package in an existing directory
add           Add dependencies to a manifest file
run, r        Run a binary or example of the local package
test, t       Run the tests
bench         Run the benchmarks
update        Update dependencies listed in Cargo.lock
search        Search registry for crates
publish       Package and upload this package to the registry
install       Install a Rust binary. Default location is
$HOME/.cargo/bin
uninstall     Uninstall a Rust binary
```

See `cargo help` for more information on a specific command.

Useful Resources

Rustlings

Rust by example

> Rust Lang Docs
> Rust Playground
>
> Rust Forum
> Rust Discord