Moving from Solidity to Cairo

*With the recent adoption of Starknet and the common talk about rollups being the "ultimate scalability tool for blockchains," development in Cairo has become a highly demanded skill in the web3 ecosystem. According to a developer* [*report*](https://www.developerreport.com/) *released by Electric Capital, the number of full-time developers in Cairo has increased by a staggering 875% over the past two years and by 83% year-on-year. Cairo is used mainly for writing smart contracts on Starknet. Since the largest number of smart contract developers come from the Ethereum ecosystem, this article aims to act as an introductory developer's guide for moving from Solidity to Cairo.*

*However, it must be noted that Cairo can freely be used outside blockchains whenever any proof of computation is required. Hence, Cairo's use will only rise in this shifting world where provable computation is becoming more popular and needed.*

## Part I. Why is Cairo so special ?

According to the Cairo [docs](https://www.cairo-lang.org/docs/), "Cairo is the first Turing-complete language for creating STARK-provable programs for general computation." The "Turing-complete" part means this programming language can simulate a Turing-complete machine. A Turing-complete (named after [Alan Turing](https://en.wikipedia.org/wiki/Alan_Turing)) machine is a computer that can perform any computation and execute any algorithm if given enough time and resources. Meanwhile, the "STARK-provable" part of the above definition underlines the property of Cairo to execute programs written in it so that it is possible to create a STARK proof of the execution efficiently. When a program written in Cairo is executed on any machine, a newly generated STARK (Scalable Transparent ARguments of Knowledge) proof allows any independent verifier to succinctly verify the honest execution of the program without having to trust the machine on which the program's computations were performed initially.

Below we will see how Cairo differs from Solidity and which language is "better" may arise. However, it may be impossible to achieve a consensus on the answer to such a question that compares "apples to oranges." Solidity is an example of the language first used for widely adopting composable computation. Composable computation is a term used to describe a programming paradigm in which different system components can be used in sync and composed into complicated software by using different components as Lego blocks. Solidity made it possible for smart contracts to be intricately intertwined due to the following:

* Byte code that was made public and transparent to the world
* Standard ABI structures (that allowed others to interact with external smart contracts)
* Use of the Ethereum network to have an almost 100% liveness (or uptime)
* Use of the Ethereum network as a means of financial transaction
* Global interoperability of smart contracts due to a single common EVM (a copy of which is held by all the validators in the network)

Even though Solidity achieved a myriad of firsts by being the most widely adopted smart contract language, in the current scenario, Cairo has a single most overpowering property: its ability to create provable programs for general computation. This single differentiating property allows us to move from a system where every computation has to be repeated hundreds of thousands of times (that's the [number of validators](https://beaconcha.in/) in the Ethereum network) to a newer system where only a single machine (called the prover) creates a proof of correct execution of a transaction that others on the network can verify. For Cairo programs' execution to be efficiently converted into a STARK proof, there are particular apparent differences between the Cairo virtual machine and the EVM. Let us go through some of the major ones.

1. The Cairo VM uses a single-write-only memory model. A memory slot cannot be overwritten (unlike the EVM). Even though this complication may seem like a highly complex overhead, the Cairo compiler solves this problem, and the abstracted version allows mutable variables to be used in code (more on this later). The single-write memory model makes Cairo a more predictable and verifiable system. Each memory address is written only once, and it retains its value until the execution of the program is complete. This means there's a clear, unchanging record of the computation, which is vital when generating STARK proofs, as they rely on verifying the correctness of computations.
2. When we talk of cryptographic proofs, we almost always speak of these proofs being created for operations on elements of a certain field. As also discussed in our article on arithmetization, in mathematics, a field is a set of elements with two binary operations, addition, and multiplication. It satisfies certain axioms, such as closure, associativity, commutativity, and the existence of inverses and identity elements. Examples of fields include real numbers, rational numbers, and complex numbers. The elements of such a field are the subjects of operations that are part of any provable general computation.

When discussing fields, it is important to note that some fields are finite (the one used in Cairo is also finite). In a finite field, the number of elements is finite, and all the elements are less than the highest "order" of the field. The "order" of a field equals the highest number that the elements of a field can attain plus one. Also, all addition and multiplication operations result in a number, the module of the highest order. So, for example, in a field of order 6, the addition 3+4 would result in (3+4) mod 6 = 7 mod 6 = 1

Cairo uses a finite field of order 2^252. Since, while generating proofs, all the elements need to be field elements, the Cairo VM uses field elements as the base for all the operations. It does not follow the conventional system of uint256 or uint32 as the EVM does. However, abstraction techniques can be created to use the more convenient uint256 type (or similar). However, these structures require more resources (increasing the number of operations proving the same in order of tens) for execution.

1. Cairo does not have the concepts of inheritance and polymorphism. Though contracts can be extended by importing specific functions and storage variables, the much-used concepts of OOP require a bit of "out-of-the-box" thinking.

When we write a Cairo smart contract, it is first converted into Sierra code, which must be published on the network. Sierra code is an abstraction over the raw Cairo assembly (CASM) code interpreted by the Cairo VM. The compilation of Cairo code to Sierra surrounds the Cairo code with some security measures, the most important of which is a mechanism to avoid DOS attacks. According to Starknet documentation, "A crucial property of every decentralized L2 is that the sequencers are guaranteed to be compensated for work they do. The notion of reverted transactions is a good example: even if the user's transaction failed mid-execution, the sequencer should be able to include it in a block and charge execution fees up to the point of failure". However, sometimes a user may write a line of code or include a transaction, proving the execution is impossible. For example, the statement `assert 1 == 0` is a valid Cairo statement; however, including this execution in a cryptographic proof is not possible since this statement is false, and it translates to polynomial constraints that are not satisfiable. Hence Sierra adds a security layer that ensures even unprovable reverted transactions are charged for. This both mitigates the potential of a DoS attack on the sequencer and satisfies the economic incentives of the sequencer.

Now that we have an idea of Cairo's basic types, functions, and structures. We will also compare those with Solidity counterparts to be able to draw more parallels between the two smart contract languages - albeit Cairo can also be used to write non-smart contract code that can be used for creating provable programs. However, that is out of our current scope of discussion.

Note that from this point onwards, all discussion regarding Cairo concerns Cairo 1. Cairo 0 is no longer recommended as the go-to language for writing smart contracts on Starknet.

**Cairo Types**

Below is the list of fundamental types in Cairo:

| u8 | unsigned 8 bit integer | uint8 |
| --- | --- | --- |
| u16, u32…u256 | unsigned 16-bit/32-bit…/256-bit integer | uint16, uint32,... uint256 |
| bool | boolean | bool |
| felt252 | field element (the most basic type that represents a “number”) | N/A |
| Option <T> | value that may or may not be present (hence “optional” type) | N/A |
| Result <T,E> | outcome of a computation that may result in an error | N/A |
| Array <T> | dynamic array structure | T[] |

As we can see in the list above, starting from Cairo 1, unsigned integers have been added to Cairo, similar to their counterparts in Solidity. Even though using integers may be less cost-effective for the sequencer than directly using felts, integrating integers into Cairo promises to simplify developers' lives.

Other than that, using Arrays is very similar to the syntax in Rust, and they are similar in logic to those in Solidity.

**Functions in Cairo**

Functions in Cairo can be of the following types:

* internal
* external
* view
* constructor

By default, all contract functions are considered internal (such functions can only be called from inside the contract - somewhat similar to private functions in other languages). "External" functions are open to the world and can be called by other smart contracts as well - including accounts (hooray, account abstraction !) - while the "view" functions are a type of external function that can only read the state on-chain. View functions cannot modify the state of the chain. "Constructor" is another attribute of functions in Cairo given to … constructors of a smart contract.

Now, let us compare the syntax of function declaration between that of Cairo and Solidity:

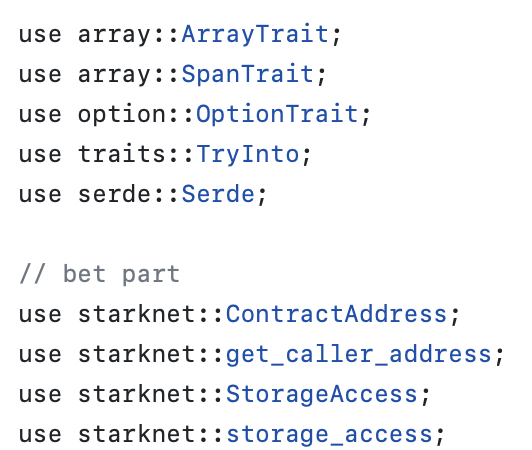
| Cairo | Solidity |
| --- | --- |
| #[view]  fn is\_cairo\_fun() -> (u8, u8) {  let yes = yes\_var::read();  let no = no\_var::read();  return (yes, no);  } | function is\_cairo\_fun() public view returns (uint8, uint8) {  uint8 yes = yes;  uint8 no = no;  return (yes, no);  } |

1. The keyword to declare a function is `fn` in Cairo, while it is `function` in Solidity.
2. The function types are declared before the function keyword in Cairo (#[view]), but in Solidity, the format is different (see above).
3. In the Cairo programming language, the syntax for declaring return values involves using the '→' symbol. On the other hand, in Solidity, the keyword 'returns' is used to denote return values.

**Modules in Cairo**

Modules in Cairo serve the purpose of grouping related functionality within a namespace. The keyword "mod" is used to define a module, followed by the module name and a code block that includes functions and other declarations. Modules can import other modules and make use of their functionality.

These are similar to libraries in other languages, and in Solidity modules can be compared to the inheritance of contracts.



For example, in the above code, `starknet` and `array` modules are imported. The syntax differs from `import` statements in Solidity or inheritance which uses the `is` keyword (see [this](https://ethereum.stackexchange.com/questions/40004/importing-other-contracts-into-one-another)). Note that the `use` keyword makes all functions from the imported module accessible within the importing module in Cairo.

**Arrays in Cairo**

Using arrays in Cairo 1.0 has become easier since manipulations of the dynamic array are made possible with the array module’s exported functions such as **append**, **array\_at**, and **array\_len.**

fn fib(num: usize) -> (Array::<felt>, felt, usize) {

let mut fibonacci\_arr = ArrayTrait::new();

fibonacci\_arr.append(1);

fibonacci\_arr.append(1);

let mut updated\_fibonacci\_arr = fib\_inner(num, fibonacci\_arr);

let length = updated\_fibonacci\_arr.len();

let last\_element = updated\_fibonacci\_arr.at(length - 1\_usize);

return (updated\_fibonacci\_arr, last\_element, length);

}

**Code walkthrough**

Now that we are introduced to the set of types and functions that can be used in Cairo, let us go over a sample ERC20 contract code [here](https://gist.github.com/MakC-Ukr/2f78dd280b5bb8a3829f07fa0169116c).

Lines 1 and 2 initialize and give the name to the smart contract  
#[contract]

mod ERC20 {

Line 4 imports `get\_caller\_address` function from `starknet` module. The get\_caller\_address() function returns the address of the contract that called the function.

use starknet::get\_caller\_address;

Line 7 to 14 define a structure (similar to `struct` in Solidity) that is used later in the code.

struct Storage {

name: felt,

symbol: felt,

decimals: u8,

total\_supply: u256,

balances: LegacyMap::<felt, u256>,

allowances: LegacyMap::<(felt, felt), u256>,

}

Lines 20 to 21 define an event Transfer that is emitted whenever the ERC20 token is transferred. Events provide an efficient way to track contract activities/state changes, thus allowing external listeners to react accordingly.

#[event]

fn Transfer(from\_: felt, to: felt, value: u256) {}

Lines 36 to 54 contain a constructor function (the function that is called when contract is deployed) and a view function that reads the `name` storage variable and returns its value to the user.

#[constructor]

fn constructor(

name\_: felt, symbol\_: felt, decimals\_: u8, initial\_supply: u256, recipient: felt

) {

name::write(name\_);

symbol::write(symbol\_);

decimals::write(decimals\_);

assert(recipient != 0, 'ERC20: mint to the 0 address');

total\_supply::write(initial\_supply);

balances::write(recipient, initial\_supply);

Transfer(0, recipient, initial\_supply);

}

// @dev get name of the token

// @return the name of the token

#[view]

fn get\_name() -> felt {

name::read()

}

Lines 97 to 101 define an external (discussed earlier) function that is used to transfer ERC20 tokens from one account to another. This function calls the `transfer\_helper()` functions underneath that also emits the Transfer() event defined above.

#[external]

fn transfer(recipient: felt, amount: u256) {

let sender = get\_caller\_address();

transfer\_helper(sender, recipient, amount);

}

If you are comfortable with Solidity, based on the above examples of code, at least a superficial level of understanding of how Starknet works, you must have gotten an idea of how to read basic Cairo smart contracts. The next step in your Cairo development journey would be to refer to the Cairo [docs](https://www.cairo-lang.org/docs/), register for [Cairo Basecamp](https://docs.google.com/forms/d/e/1FAIpQLSf2k9vjPpeymbUpJMRDuN3QqNcHtjWx8whX2wY4EbihF1EaPg/viewform), or go through the tutorials [here](https://www.starknet.io/en/tutorials). To get a demo of setting up the environment for local compilation and deployment of contracts, refer to the video [here](https://www.youtube.com/watch?v=jERD2vH1hOY).

**Sources**

1. <https://www.youtube.com/watch?v=uW2tTR53J7o>
2. <https://medium.com/yagi-fi/provable-vs-composable-computation-or-why-cairo-will-supersede-solidity-6b00e69bfc9e>
3. <https://github.com/starknet-edu/starknetbook>
4. <https://www.argent.xyz/blog/getting-started-with-cairo-1.0/>