5 top practices for beginning Cairo devs

**Introduction**

Starknet is an validity rollup built on Ethereum, an L2 that promises to deliver unmatched scalability and robust security .Starknet batches multiple transactions into chunks and periodically submits a validity proof for these chunks on Ethereum. As it submits proofs of validity for large amounts of transactions, it significantly reduces congestion and transaction costs on Ethereum mainnet, proving itself to be a game-changer in the world of blockchain.These cryptographic proofs submitted by the Starknet sequencer are secured by a staggering 700,000 unique validators on Ethereum.

Integral to building applications on Starknet (which are the source of the huge number of transactions) is the novel smart contract programming language - Cairo. Cairo is a Rust-inspired language that allows developers to write Starknet smart contracts in a safe and convenient manner - it also allows writing provable programs, in general.

If you're well-versed in Rust, that's a substantial plus when you turn to Cairo for writing smart contracts. The likeness is striking - Rust's mod keyword, use statements, struct keywords, attribute-like annotations, function definitions, and let bindings all find echoes in Cairo's syntax.

The similarity of Rust and Cairo can be seen in the below given snippet of [code](https://github.com/keep-starknet-strange/alexandria/blob/main/src/data_structures/queue.cairo) implementing queues in Starknet:

// Core lib imports

use array::ArrayTrait;

const ZERO\_USIZE: usize = 0;

struct Queue<T> {

elements: Array<T>,

}

trait QueueTrait<T> {

fn new() -> Queue<T>;

fn enqueue(ref self: Queue<T>, value: T);

fn dequeue(ref self: Queue<T>) -> Option<T>;

fn peek\_front(self: @Queue<T>) -> Option<Box<@T>>;

fn len(self: @Queue<T>) -> usize;

fn is\_empty(self: @Queue<T>) -> bool;

}

Shifting focus, Solidity know-how is another major advantage. The shared ethos of a security-first approach, and awareness of potential pitfalls such as overflows, are concepts that transition smoothly from Solidity to Cairo. Both expertise areas contribute uniquely to refining smart contract development in Cairo. For Solidity devs learning Cairo, there is a useful introductory article comparing the two of the most powerful smart contract languages [here](https://medium.com/starkware/moving-from-solidity-to-cairo-7d44f9723c68).

**5 Top Practises for Writing Code in Cairo**

1. **Security-first approach**

The immutable and public nature of blockchains is a very important fact that any smart contract developer keeps in mind when writing on-chain applications. For public blockchains (and, consequently, transparent layers such as Starknet that are built on top of them) a “security-first” approach of writing smart contracts is non-negotiable and must be kept in mind from the start.

Whenever a smart contract developer starts writing code for blockchain applications, first of all, an approximate structure must be thought of for the various relations between smart contracts and function accessibility.

* **Prevent Overflows and Underflows**: Be aware of the maximum and minimum values for your data types, and ensure that your code can handle overflow and underflow scenarios gracefully. Consider using verified libraries or built-in language features to manage these issues (since Cairo is a rapidly-evolving language right now, usage of libraries would also be done with caution). Some popular Cairo libraries are [Openzeppelin](https://github.com/OpenZeppelin/cairo-contracts), [Cairo Common](https://perama-v.github.io/cairo/cairo-common-library/).
* **Validate Inputs and Outputs**: Never trust user-supplied input. Always validate and sanitize inputs to prevent attacks such as injection or logic errors. Similarly, validate outputs to ensure they are within the expected limits.
* **Robust Error Handling**: Whenever error or unexpected values are encountered, the code execution must fail and a relevant error message be emitted for the user. With Cairo 1.0 reverted transactions are also included in the block, so a relevant error message ensures transparency and usability of on-chain applications.
* **Beware of Reentrancy Attacks:** Be cautious with external calls and consider using mechanisms like mutexes or the Checks-Effects-Interactions pattern to prevent reentrancy attacks.

The above mentioned guidelines merely scratch the surface of secure smart contract programming in Cairo. It's important to understand that there's a plethora of other preventative steps vital for creating secure applications on-chain. This is particularly crucial as one of the most notable features of blockchains - transparency - can easily become a “double-edged sword”, offering visibility but also exposing vulnerabilities to potential attackers.

1. **Testing**

Tests are Cairo functions that verify that the rest of the code functions as expected by the developers. Writing tests for logic of your program is an essential part of the complete programming process. Given the high stakes involved - with smart contracts often responsible for large volumes of transactions and consequently, significant sums of money - comprehensive testing is not just important, it's absolutely vital. Unit tests for smart contracts can be written with tools like [pytest](https://docs.pytest.org/en/7.4.x/), [Protostar](https://docs.swmansion.com/protostar/) and [Starknet Foundry](https://github.com/foundry-rs/starknet-foundry).

Setting up testing in Starknet Foundry is extremely easy, since when a project is set up in Protostar, tests can directly be added and run with the `*snforge*` command. A very simple test example written for Starknet Foundry is given below:

fn sum(a: felt252, b: felt252) -> felt252 {

return a + b;

}

#[test]

fn test\_sum() {

assert(sum(2, 3) == 5, 'sum incorrect');

}

Writing tests align with development of the smart contracts ensures that your code has less bugs and is more compact in logic. Note that testing does not ensure a bug-free code but drastically reduces chances of error, and makes them more easily spottable in case of an audit before production.

1. **Gas efficiency**

Starknet can process thousands of transactions in a day and didn’t we state in the beginning of the article that transactions are much cheaper than Ethereum mainnet. Nevertheless, one of the most important practices is still writing gas efficient code ? Writing gas-efficient code is important not only due to cost-efficiency; it is a matter of network health as well. An ideal scenario where transaction costs are virtually free opens up blockchains to a lto more new possibilities but also makes the network an easy target for DDoS (Distributed Denial of Service) attacks. Additionally, miners, sequencers, and validators always need incentives to keep operating so transaction fees can never become virtually free (this doesn’t mean that they won’t become cheap enough for daily spendings to occur on-chain).

In such a world, writing code that takes into account gas efficiency makes sense from a long-term perspective. If the applications built by the developers consume a lto of gas, users will start avoiding the product and new competitive projects would be incentivised to challenge your product. Moreover, in transaction-heavy industries such as gaming, cost savings can significantly aggregate over time.

1. **Modular Cairo programming**

As you start writing large programs, managing very long smart contracts becomes difficult and prone to error. In order to avoid having a small number of contract codes that are extremely large in size, code of similar functionality can be grouped. This also makes reusing code and adding more functionality in future simpler and cleaner.

In Cairo, in order to break down a single large file of code, you can split it into multiple modules and then into multiple files. Splitting your code into modules is similar to how files on your PC are located in separate directories. Modules can also be defined inside other modules and you can access functions from other modules in order to re-use code more easily.

Below is an example of modules you would define in an airline management program.

mod check\_in\_counter {

mod passengers {

fn check\_in() {}

fn issue\_boarding\_pass() {}

}

mod baggage {

fn check\_baggage() {}

fn issue\_baggage\_receipt() {}

}

}

fn fly\_with\_airline() {

// Accessing functions from other modules by absolute path

airline::check\_in\_counter::passengers::check\_in();

// Accessing functions from other modules by relative path

check\_in\_counter::passengers::check\_in(); //

}

The above example also showcases how easily functions from foreign modules can be imported and called.

1. **Use the power of Starknet**

This “top practise” is not specifically a programming advice but rather a suggestion to build new kinds of applications that are made possible due to networks like Starknet.

Certainly, here's a possible transition paragraph:

As we move forward, it's vital to remember that the potential of technologies like Starknet isn't limited to just reproducing existing blockchain applications. Some fields such as DeFi (Decentralized Exchanges, options markets) and NFTs have bloomed on other blockchains and have proven their value in the blockchain ecosystem. Building such applications is a necessary step in the ladder of growth for Starknet as an ecosystem.

However, the true power of Starknet shall also be unleashed by building novel architectures and protocols that are made possible due to the transparent and scalable technology of validity proofs (Starknet). Starknet’s high TPS and cheap transactions could revolutionize future of gaming and make look very different. Similarly,