Unlocking StarkNet with Cairo - the *otter* way

Author: Pintea, Tudor

"Unlocking StarkNet with Cairo " is an essential resource for developers venturing into the realm of StarkNet, the permissionless decentralized ZK-Rollup operating on the Ethereum blockchain.

*note: if you find this cookbook useful or you think I m a capable person of being a Starknet Delegate you can endorse me here ---> https://delegate.starknet.io/profile/61425.eth#overview

made with ♡ by @tudorpintea999

This comprehensive guide spans 100 pages, meticulously crafted to demystify the Cairo programming language and StarkNet's distinctive ecosystem. Through a curated collection of practical examples, readers will master the art of creating robust and efficient smart contracts tailored for a variety of applications – from basic token transfers to complex DeFi protocols. Each example is designed to build on the last, forming a solid foundation of knowledge that culminates in the ability to craft advanced contracts for real-world use cases. Whether you're a novice eager to step into blockchain development or a seasoned programmer looking to expand your toolkit, this book offers the insights and hands-on experience needed to harness the power of StarkNet and revolutionize the blockchain space.

Creating a simple smart contract in Cairo, the programming language specifically designed for StarkNet, involves several steps. Let's start with a basic example: a smart contract that stores and updates a simple integer value.

1. Simple Smart Contract

Setting Up the Environment

Before you begin writing your smart contract, ensure that you have the necessary development environment set up. This typically includes:

- Cairo language installation.

- A development environment like an IDE (Integrated Development Environment) that supports Cairo.

- Access to a testnet for deploying and testing your contract.

Writing the Smart Contract

Here's a basic example of a Cairo smart contract:

```cairo
%lang starknet

from starkware.cairo.common.cairo_builtins import HashBuiltin

@storage_var

func data() -> (value : felt):

end

@external

func initialize():

    data.write(value=0)

    return ()

end

@external

func update_data(new_value : felt):

    data.write(value=new_value)

    return ()

end

@view

func get_data() -> (value : felt):

    let (value) = data.read()
```

```
    return (value)

end

```

Explanation of the Code

- **`%lang starknet`**: This line indicates that the file is a StarkNet contract written in Cairo.

- **`@storage_var`**: This decorator defines a storage variable, `data`, which stores a single field element (`felt`).

- **`@external`**: Functions with this decorator can be called externally. Here, `initialize` sets the initial value of `data` to 0, and `update_data` allows updating its value.

- **`@view`**: This decorator is used for functions that read data but don't modify it. The `get_data` function returns the current value of `data`.

Deploying and Testing

After writing the contract, you would typically compile it using the Cairo compiler and deploy it to the StarkNet testnet. You can then interact with the contract using StarkNet tools, sending transactions to call `initialize`, `update_data`, and `get_data` functions.

Next Steps

This example is a starting point. As you get more familiar with Cairo, you can expand your contracts to include more complex logic, interact with other contracts, and handle more intricate data structures.

Remember, Cairo and StarkNet are constantly evolving, so it's crucial to stay updated with the latest documentation and community best practices.

2. $TOFU token smart contract

The features of $TOFU include:

- A fixed supply of 21,000,000 tokens.

- 18 decimal places.

- Burnable tokens.

- Non-mintable after initial creation.

- Ability to pause transactions.

- Ownership transfer capability.

- 2% transaction tax.

- Airdrop functionality.

We will outline a basic structure of such a contract. Note that this example is simplified and should not be used in a production environment without thorough testing and security audits. Also, Cairo's syntax and capabilities may evolve, so it's essential to consult the latest documentation.

Basic Structure of the Token Contract in Cairo

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin


@storage_var
func total_supply() -> (supply : felt):
end


@storage_var
func balance_of(account : felt) -> (balance : felt):
end


@storage_var
func owner() -> (owner : felt):
end


@storage_var
func is_paused() -> (paused : felt):
end
```

```
@view
func get_total_supply() -> (supply : felt):
    let (supply) = total_supply.read()
    return (supply)
end


@external
func initialize(initial_owner : felt):
    total_supply.write(21000000 * (10 ** 18))  # Total supply with 18
decimal places
    balance_of.write(initial_owner, 21000000 * (10 ** 18))  # Assign all
tokens to initial owner
    owner.write(initial_owner)
    is_paused.write(0)
    return ()
end


@external
func transfer(sender : felt, receiver : felt, amount : felt):
    func assert_not_paused():
    let (paused) = is_paused.read()
```

```
    assert paused == 0, 'Contract is paused'

    return ()

end

    assert is_owner(sender), 'Only owner can transfer tokens'


    let (sender_balance) = balance_of.read(sender)

    let (receiver_balance) = balance_of.read(receiver)


    let tax = amount * 2 / 100  # 2% tax

    let net_amount = amount - tax


    balance_of.write(sender, sender_balance - amount)

    balance_of.write(receiver, receiver_balance + net_amount)

    # Optionally distribute or burn the tax amount


    return ()

end


@external

func burn(account : felt, amount : felt):

    assert_not_paused()
```

```
    assert is_owner(account), 'Only owner can burn tokens'


    let (account_balance) = balance_of.read(account)
    balance_of.write(account, account_balance - amount)


    let (current_supply) = total_supply.read()
    total_supply.write(current_supply - amount)
    return ()
end


@external
func pause():
    assert is_owner(caller), 'Only owner can pause'
    is_paused.write(1)
    return ()
end


@external
func unpause():
    assert is_owner(caller), 'Only owner can unpause'
    is_paused.write(0)
```

```
    return ()
end


@external
func transfer_ownership(new_owner : felt):
    assert is_owner(caller), 'Only owner can transfer ownership'
    owner.write(new_owner)
    return ()
end


@external
func airdrop(recipients : felt*, amounts : felt*, num_recipients :
felt):
    assert_not_paused()
    assert is_owner(caller), 'Only owner can airdrop'


    for i in range(num_recipients):
        let recipient = recipients[i]
        let amount = amounts[i]


        let (recipient_balance) = balance_of.read(recipient)
```

```
        balance_of.write(recipient, recipient_balance + amount)


    return ()
end


func is_owner(account : felt) -> (is_owner : felt):
    let (current_owner) = owner.read()
    return account == current_owner
end


func assert_not_paused():
    let (paused) = is_paused.read()
    assert paused == 0, 'Contract is paused'
    return ()
end
```

The only_owner function ensures that only the contract owner can call certain functions, like pausing or unpausing the contract.

```
cairo
func only_owner():
```

```
    let (current_owner) = owner.read()

    assert syscall_ptr.get_caller_address() == current_owner, 'Only the
owner can call this function'

    return ()

end
```

Key Points:

- **Initialize Function**: Sets the total supply, assigns all tokens to the initial owner, and sets the contract state.

- **Transfer Function**: Includes a 2% tax on transactions, which can be burned or redistributed.

- **Burn Function**: Allows the owner to burn tokens, reducing the total supply.

- **Pause/Unpause Functions**: Enable or disable token transfers.

- **Ownership Transfer**: Allows changing the contract's owner.

- **Airdrop Function**: Distributes tokens to multiple addresses.

Testing and Deployment

This contract needs to be thoroughly tested, especially for edge cases, security vulnerabilities (like reentrancy, overflow/underflow, etc.), and compliance with your project's requirements. After testing, you can compile and deploy it to the StarkNet testnet or mainnet using appropriate tools and procedures.

3. Token swap on Starknet example

Creating a smart contract for token swapping on the StarkNet platform involves several critical components. This kind of contract typically allows users to exchange one type of token for another, based on predefined exchange rates or market dynamics. Given the complexity of such a contract, I will provide a high-level outline and basic structure for a simple token swap contract.

Basic Structure of a Token Swap Contract

This contract assumes two types of tokens (Token A and Token B) are available for swapping. The contract will manage the exchange rate and facilitate the swap between these tokens.

Cairo Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for tracking balances, exchange rates, and contract ownership.

```cairo
@storage_var
func balance_of(token: felt, account: felt) -> (balance: felt):
end


@storage_var
func exchange_rate(token_a_to_b: felt) -> (rate: felt):
end


@storage_var
func owner() -> (owner: felt):
end
```

Initialization

Set the initial exchange rate and contract owner.

```cairo
@external
```

```
func initialize(contract_owner: felt, initial_rate: felt):

    owner.write(contract_owner)

    exchange_rate.write(initial_rate)

    return ()

end

```

Token Swap Function

Implement a function to swap tokens. This requires updating balances and considering the exchange rate.

```cairo

@external

func swap_tokens(sender: felt, token_from: felt, token_to: felt, amount: felt):

    assert_not_paused()

    assert exchange_rate.exists(), 'Exchange rate not set'



    let (rate) = exchange_rate.read()

    let amount_to_receive = amount * rate



    let (sender_balance_from) = balance_of.read(token_from, sender)
```

```
    let (sender_balance_to) = balance_of.read(token_to, sender)


    assert sender_balance_from >= amount, 'Insufficient balance'


    balance_of.write(token_from, sender, sender_balance_from - amount)

    balance_of.write(token_to, sender, sender_balance_to +
amount_to_receive)


    emit SwapEvent(sender, token_from, token_to, amount,
amount_to_receive)

    return ()

end

```

Helper Functions

Implement functions for ownership verification and pausing the contract.

```cairo

func is_owner(account: felt) -> (is_owner: felt):

    let (current_owner) = owner.read()

    return account == current_owner

end

```

```
func assert_not_paused():
    let (paused) = is_paused.read()
    assert paused == 0, 'Contract is paused'
    return ()
end
```

Event Emission

Define events for logging contract activity.

```cairo
@event
func SwapEvent(sender: felt, token_from: felt, token_to: felt,
amount_sent: felt, amount_received: felt):
end
```

Considerations and Next Steps

1. **Exchange Rate Management**: The contract should include mechanisms to update exchange rates, either manually by the owner or through an oracle for market-based rates.

2. **Liquidity and Token Balances**: The contract needs a mechanism to manage liquidity for each token type. This could involve deposit and withdrawal functions for the contract owner or liquidity providers.

3. **Security and Efficiency**: Ensure the contract is secure against common vulnerabilities (e.g., reentrancy, integer overflow/underflow). Optimize gas usage to make operations cost-effective.

4. **Testing and Deployment**: Thoroughly test the contract, especially focusing on edge cases and security aspects. After testing, deploy it to a testnet for further testing before moving to the mainnet.

5. **Compliance with Standards**: Ensure that the contract complies with StarkNet's standards and practices.

6. **Frontend Integration**: For user interaction, a frontend application that interacts with this contract will be necessary.

7. **Professional Review**: Due to the complexity and potential financial risks, it is highly recommended to have this contract reviewed by experienced blockchain developers or auditors.

This outline provides a basic framework. Developing a fully functional and secure token swap contract for StarkNet would require extensive development, testing, and expertise in Cairo and blockchain security.

4. Liquidity pool example

Creating a smart contract in Cairo for managing liquidity in a pool, including functions for depositing, withdrawing liquidity, withdrawing rewards, and withdrawing a specific amount, requires careful consideration of the contract's structure and logic. Below, I'll

outline a basic structure for such a contract. Please note that this is a simplified example and should be thoroughly tested and audited before any real use.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for balances, rewards, and liquidity tracking.

```cairo
@storage_var
func user_balance(user: felt) -> (balance: felt):
end


@storage_var
func user_rewards(user: felt) -> (reward: felt):
end
```

```
@storage_var

func total_liquidity() -> (liquidity: felt):

end
```

Deposit Tokens to Liquidity Pool

Function to deposit tokens into the liquidity pool.

```
```cairo

@external

func deposit_liquidity(user: felt, amount: felt):

    let (current_balance) = user_balance.read(user)

    user_balance.write(user, current_balance + amount)



    let (current_liquidity) = total_liquidity.read()

    total_liquidity.write(current_liquidity + amount)

    return ()

end

```
```

Withdraw Liquidity

Function to withdraw a specific amount of liquidity.

```cairo
@external
func withdraw_liquidity(user: felt, amount: felt):
    let (current_balance) = user_balance.read(user)
    assert current_balance >= amount, 'Insufficient balance'

    user_balance.write(user, current_balance - amount)

    let (current_liquidity) = total_liquidity.read()
    total_liquidity.write(current_liquidity - amount)
    return ()
end
```

Withdraw Rewards

Function to withdraw only the rewards.

```cairo
@external
func withdraw_rewards(user: felt):
    let (current_reward) = user_rewards.read(user)
    assert current_reward > 0, 'No rewards to withdraw'
```

```
    # Logic to transfer rewards to the user
    # Reset the rewards after withdrawal
    user_rewards.write(user, 0)
    return ()
end
```

Withdraw All Deposits

Function to withdraw all deposits and rewards.

```cairo
@external
func withdraw_all(user: felt):
    let (current_balance) = user_balance.read(user)
    let (current_reward) = user_rewards.read(user)

    # Logic to transfer total balance and rewards to the user
    user_balance.write(user, 0)
    user_rewards.write(user, 0)

    let (current_liquidity) = total_liquidity.read()
```

```
    total_liquidity.write(current_liquidity - current_balance)

    return ()

end

```

Reward Calculation

Implement logic to calculate and update rewards over time. This could be a separate function or part of other functions like deposit or withdrawal.

Security and Functionality Considerations

1. **Token Interaction**: The contract does not currently interact with an actual token contract. Integrating with a token contract (e.g., ERC-20) is necessary for handling real tokens.

2. **Reward Distribution**: Implement a robust mechanism for calculating and distributing rewards based on the user's share of the liquidity pool.

3. **Reentrancy Guard**: Protect functions against reentrancy attacks.

4. **Audit and Testing**: Conduct thorough testing, especially for functions that handle assets. An audit by a professional is strongly recommended.

5. **Gas Optimization:** Optimize the contract for gas efficiency, particularly for functions that are called frequently.

6. **Interface and Frontend**: For user interaction, create a user-friendly interface that interacts with this contract.

This outline provides a basic structure for a liquidity management smart contract on StarkNet. It requires further development, testing, and security measures to be suitable for real-world deployment.

5. NFT example

Creating a smart contract for Non-Fungible Tokens (NFTs) involves several core functionalities like minting, transferring, and tracking ownership of unique tokens. Below, I'll provide a basic structure for an NFT smart contract in Cairo, the language used on StarkNet. This example includes functionalities for minting NFTs with unique identifiers and transferring ownership.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for NFT ownership, total supply, and maximum supply.

```cairo
```

```
@storage_var

func nft_owner(token_id: felt) -> (owner: felt):

end



@storage_var

func total_supply() -> (supply: felt):

end



const MAX_SUPPLY = 10000  # Maximum number of NFTs that can be minted

```

Minting NFTs

Function to mint a new NFT. Each NFT should have a unique identifier.

```cairo
@external

func mint_nft(to: felt, token_id: felt):

    assert token_id_is_valid(token_id), 'Invalid token ID'

    assert nft_owner.read(token_id).owner == 0, 'Token already minted'



    let (current_supply) = total_supply.read()

    assert current_supply < MAX_SUPPLY, 'Max supply reached'
```

```
    nft_owner.write(token_id, to)
    total_supply.write(current_supply + 1)
    return ()
end
```

Transferring NFTs

Function to transfer ownership of an NFT.

```
```cairo
@external
func transfer_nft(from: felt, to: felt, token_id: felt):
    assert nft_owner.read(token_id).owner == from, 'Not the owner'
    nft_owner.write(token_id, to)
    return ()
end
```
```

Helper Functions

Include helper functions for validation and other repeated logic.

```
```cairo
func token_id_is_valid(token_id: felt) -> (is_valid: felt):
```
```

```
    # Add logic to validate token_id if needed

    return (1)

end

```

Event Logging

Define events for logging activities like minting and transferring NFTs.

```cairo
@event

func MintEvent(to: felt, token_id: felt):

end



@event

func TransferEvent(from: felt, to: felt, token_id: felt):

end

```

Key Points and Additional Features

1. **Unique Identifiers**: Ensure each NFT has a unique identifier (token_id).

2. **Ownership Management**: Properly manage ownership transfers to prevent unauthorized actions.

3. **Minting Restrictions**: Implement checks to control who can mint new NFTs (e.g., contract owner or designated minter).

4. **Max Supply**: Enforce the maximum supply of NFTs that can be minted.

5. **Event Emissions**: Emit events for minting and transferring NFTs for off-chain tracking and verification.

6. **Metadata Management**: Consider how to handle NFT metadata. This often involves off-chain storage (like IPFS) with on-chain references.

7. **Security**: Implement security checks, such as reentrancy guards and validation of inputs.

Testing and Deployment

Thoroughly test the contract to ensure it behaves correctly under various scenarios and adheres to security best practices. After testing, deploy it to a testnet for further testing before considering a mainnet launch.

Final Note

This basic structure provides a starting point for an NFT smart contract. The real-world implementation of NFT contracts can be more complex, especially when considering

metadata management, compliance with standards (like ERC-721 or ERC-1155), and integration with marketplaces or other platforms.

6. Basic Structure of a Voting System Smart Contract

Developing a decentralized voting system as a smart contract involves creating a transparent and tamper-proof mechanism for casting and counting votes. Below is a basic structure for such a voting system smart contract in Cairo, the language used for StarkNet. This contract allows users to vote on different proposals, ensuring transparency and immutability of votes.

Contract Setup

```cairo
%lang starknet

from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for tracking votes, proposals, and voter registration.

```cairo
@storage_var
```

```
func proposals(proposal_id: felt) -> (description: felt, vote_count:
felt):

end

@storage_var

func votes(proposal_id: felt, voter: felt) -> (voted: felt):

end

@storage_var

func is_voter_registered(voter: felt) -> (registered: felt):

end

```

Proposal Management

Functions to create and manage proposals.

```cairo

@external

func create_proposal(proposal_id: felt, description: felt):

    # Ensure proposal_id is unique

    assert proposals.read(proposal_id).description == 0, 'Proposal
already exists'

    proposals.write(proposal_id, description, 0)

    return ()
```

```
end
```

Voter Registration

Function to register voters. This could be open to all or restricted.

```cairo
@external
func register_voter(voter: felt):
    # Optionally add logic to restrict who can register
    is_voter_registered.write(voter, 1)
    return ()
end
```

Voting Function

Function to cast a vote for a proposal.

```cairo
@external
func vote(proposal_id: felt, voter: felt):
```

```
    assert is_voter_registered.read(voter).registered == 1, 'Voter not
registered'

    assert votes.read(proposal_id, voter).voted == 0, 'Already voted'



    let (description, vote_count) = proposals.read(proposal_id)

    assert description != 0, 'Proposal does not exist'



    votes.write(proposal_id, voter, 1)

    proposals.write(proposal_id, description, vote_count + 1)

    return ()

end
```

Viewing Results

Function to view the current vote count for a proposal.

```cairo
@view

func get_vote_count(proposal_id: felt) -> (vote_count: felt):

    let (description, vote_count) = proposals.read(proposal_id)

    return (vote_count)

end
```

Event Logging

Define events for logging activities like proposal creation and voting.

```cairo
@event
func ProposalCreated(proposal_id: felt, description: felt):
end


@event
func VoteCasted(voter: felt, proposal_id: felt):
end
```

Key Points and Additional Features

1. **Proposal Uniqueness**: Ensure that each proposal has a unique identifier.

2. **Voter Eligibility**: Implement a mechanism to determine who is eligible to vote. This can be open to all or restricted based on certain criteria.

3. **Double Voting Prevention**: Ensure that a voter can only vote once per proposal.

4. **Transparency**: Maintain transparency in vote counting and proposal management.

5. **Security**: Implement security checks, such as validation of inputs and prevention of unauthorized proposal creation or voting.

6. **Vote Delegation**: Optionally, you can add a feature for vote delegation, where a voter can delegate their voting power to another voter.

7. **Voting Period**: Implement a mechanism to define a start and end time for voting on each proposal.

Testing and Deployment

Thoroughly test the contract to ensure that it functions correctly and securely under various scenarios. After testing, deploy it to a testnet for further validation before considering a mainnet launch.

Final Note

This basic structure provides a starting point for a decentralized voting system smart contract. Real-world implementations might require additional features, such as more complex voter eligibility criteria, proposal categorization, and integration with a user-friendly interface for ease of use.

7. Basic Structure of a DAO Smart Contract

Creating a smart contract for a Decentralized Autonomous Organization (DAO) involves developing a system where token holders can propose, vote, and implement decisions. This contract will include functionalities for proposal creation, voting based on token holdings, and executing decisions based on the outcomes of votes. Below is a high-level outline of how such a contract might look in Cairo, used on StarkNet.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for proposals, votes, and token balances.

```cairo
@storage_var
func proposals(proposal_id: felt) -> (description: felt, vote_count:
felt, active: felt):
end


@storage_var
func votes(proposal_id: felt, voter: felt) -> (vote_weight: felt):
end


@storage_var
func token_balance(holder: felt) -> (balance: felt):
```

```
end
```

Proposal Creation

Function for token holders to create new proposals.

```cairo
@external
func create_proposal(proposal_id: felt, description: felt):
    assert proposals.read(proposal_id).description == 0, 'Proposal already exists'
    proposals.write(proposal_id, description, 0, 1)  # Mark as active
    return ()
end
```

Voting on Proposals

Function for token holders to vote on proposals. Votes are weighted by token balance.

```cairo
@external
func vote_on_proposal(proposal_id: felt, voter: felt, support: felt):
    let (description, vote_count, active) = proposals.read(proposal_id)
```

```
    assert active == 1, 'Proposal not active'

    assert description != 0, 'Proposal does not exist'



    let (voter_balance) = token_balance.read(voter)

    assert voter_balance > 0, 'No voting power'



    let (voter_vote_weight) = votes.read(proposal_id, voter)

    assert voter_vote_weight == 0, 'Already voted'



    let updated_vote_count = vote_count + (voter_balance if support == 1
else 0)

    votes.write(proposal_id, voter, voter_balance)

    proposals.write(proposal_id, description, updated_vote_count,
active)

    return ()

end
```

Executing Proposals

Function to execute a proposal after voting is concluded. This would typically be a separate process, perhaps involving a multisig wallet or a specific execution logic.

```
```cairo

@external

func execute_proposal(proposal_id: felt):

```
```

```
    let (description, vote_count, active) = proposals.read(proposal_id)

    assert active == 1, 'Proposal not active or already executed'



    # Implement execution logic based on vote_count and specific rules



    proposals.write(proposal_id, description, vote_count, 0)  # Mark as
executed

    return ()

end

```

Token Balance Management

Functions to manage token balances, which are used for voting.

```cairo

@external

func update_token_balance(holder: felt, balance: felt):

    # Implement logic for updating token balances, possibly linked to an
ERC-20 token contract

    token_balance.write(holder, balance)

    return ()

end
```

```

## Event Logging

Define events for logging activities like proposal creation, voting, and execution.

```cairo
@event
func ProposalCreated(proposal_id: felt, description: felt):
end

@event
func VoteCasted(voter: felt, proposal_id: felt, vote_weight: felt):
end

@event
func ProposalExecuted(proposal_id: felt):
end
```

Considerations and Enhancements
```

1. **Proposal Validation**: Implement additional checks to validate proposals based on DAO rules.

2. **Voting Power**: The voting power of each member could be determined by the number of governance tokens they hold.

3. **Token Integration**: Integrate with a token contract (e.g., an ERC-20 contract on StarkNet) to manage token balances.

4. **Proposal Execution**: Define a clear mechanism for executing decisions made by the DAO. This could involve smart contract calls, treasury management, etc.

5. **Security Measures**: Include security measures like reentrancy guards, and consider using a multisig approach for critical operations.

6. **Governance Rules**: Clearly define and implement the governance rules, including how proposals are approved (e.g., majority vote, quorum requirements).

7. **Timelocks and Delays**: Implement timelocks or delays for executing proposals after approval for additional security and transparency.

Testing and Deployment

Extensive testing is crucial to ensure the contract functions as expected and is secure. Deploy the contract to a testnet for thorough testing before considering a mainnet launch.

Conclusion

This outline provides a foundational structure for a DAO smart contract. Depending on the specific requirements and governance model of the DAO, additional features and complexities may need to be added. Remember that DAO contracts often manage significant assets and should be developed with utmost care and thorough security auditing.

8. Basic Structure of a Staking Smart Contract

To develop a smart contract for a staking mechanism in Cairo for StarkNet, we would need to consider the key functionalities such as staking tokens, calculating rewards over time, and allowing users to withdraw their stake and rewards. Below, I will outline a basic structure for a staking contract.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for staked balances, reward balances, and total staked amount.

```cairo
@storage_var
func staked_balance(user: felt) -> (amount: felt):
```

```
end


@storage_var
func reward_balance(user: felt) -> (amount: felt):
end


@storage_var
func total_staked() -> (amount: felt):
end
```

Staking Tokens

Function for users to stake their tokens.

```cairo
@external
func stake_tokens(user: felt, amount: felt):
    # Update the user's staked balance
    let (current_balance) = staked_balance.read(user)
    staked_balance.write(user, current_balance + amount)

    # Update the total staked amount
```

```
    let (current_total) = total_staked.read()

    total_staked.write(current_total + amount)

    return ()

end
```

Calculating Rewards

Function to calculate rewards, which could be called periodically or upon certain actions like staking/unstaking.

```cairo
func calculate_rewards(user: felt):

    # Define reward calculation logic, which might include

    # factors like staking duration and amount

    let (staked_amount) = staked_balance.read(user)

    let rewards = staked_amount * REWARD_RATE  # Simplified reward calculation

    reward_balance.write(user, rewards)

    return ()

end
```

Withdrawing Staked Tokens and Rewards

Function to withdraw staked tokens and rewards.

```cairo
@external
func withdraw_stake_and_rewards(user: felt):
    let (staked_amount) = staked_balance.read(user)
    let (earned_rewards) = reward_balance.read(user)

    # Implement token transfer logic here
    # For example, calling an ERC-20 contract's transfer function

    # Reset user's staked balance and rewards
    staked_balance.write(user, 0)
    reward_balance.write(user, 0)

    # Update the total staked amount
    let (current_total) = total_staked.read()
    total_staked.write(current_total - staked_amount)
    return ()
end
```

Withdraw Partial Stake

Function to withdraw a partial amount of the staked tokens.

```cairo
@external
func withdraw_partial_stake(user: felt, amount: felt):
    let (staked_amount) = staked_balance.read(user)
    assert staked_amount >= amount, 'Insufficient staked balance'

    # Adjust the user's staked balance and total staked amount
    staked_balance.write(user, staked_amount - amount)
    let (current_total) = total_staked.read()
    total_staked.write(current_total - amount)

    # Implement token transfer logic here

    return ()
end
```

Event Logging

Define events for logging activities like staking and withdrawal.

```cairo
@event
func StakeEvent(user: felt, amount: felt):
end

@event
func WithdrawalEvent(user: felt, amount: felt):
end
```

Considerations and Enhancements

1. **Token Contract Integration**: The staking contract should interact with a token contract to handle the actual transfer of tokens when staking and withdrawing.

2. **Reward Rate**: Define a constant or variable `REWARD_RATE` based on your staking model.

3. **Reward Calculation**: The `calculate_rewards` function should implement a more complex reward calculation mechanism based on various factors such as staking duration or amount.

4. **Lockup Period**: Optionally add a lockup period during which staked tokens cannot be withdrawn.

5. **Slashing**: If applicable, implement a slashing mechanism for stakers who perform undesirable actions.

6. **Security**: Implement security best practices, such as checks-effects-interactions pattern, to avoid reentrancy and other vulnerabilities.

7. **Gas Optimization**: Optimize for gas efficiency, as staking contracts are often used frequently.

8. **Testing and Auditing**: Extensively test the contract for correctness and security, and conduct an audit before deploying to mainnet.

This basic structure provides a foundation for a staking smart contract. Real-world implementation would require a more detailed approach, especially in handling rewards distribution and ensuring secure interactions with token contracts.

9. Basic Structure of a Multi-Signature Wallet Smart Contract

Creating a multi-signature wallet smart contract involves establishing a method where a transaction requires multiple distinct approvals before execution. This kind of contract is especially useful for managing shared funds in a secure manner. Here's a conceptual outline of how such a contract might look in Cairo for StarkNet:

Contract Setup

```cairo
```

```
%lang starknet



from starkware.cairo.common.cairo_builtins import HashBuiltin

```

Storage Variables

Define storage variables to keep track of owners, the number of required confirmations, and pending transactions.

```cairo
@storage_var

func is_owner(address: felt) -> (is_owner: felt):

end



@storage_var

func transaction_count() -> (count: felt):

end



@storage_var

func transaction_details(tx_id: felt) -> (to: felt, value: felt, data:
felt, executed: felt):
```

```
end


@storage_var
func confirmations(tx_id: felt, owner: felt) -> (confirmed: felt):
end


@storage_var
func required_confirmations() -> (count: felt):
end
```

Contract Initialization

Initialize the contract with the addresses of the owners and the number of required confirmations.

```cairo
@external
func initialize(owners: felt*, required: felt):
    # Set the required number of confirmations
    required_confirmations.write(required)

    # Register the owners
```

```
    let owners_count = len(owners)

    for i in range(owners_count):

        is_owner.write(owners[i], 1)

    return ()

end
```

Submitting Transactions

Function to submit a new transaction proposal.

```cairo
@external

func submit_transaction(to: felt, value: felt, data: felt):

    only_owner()


    let (tx_index) = transaction_count.read()

    transaction_details.write(tx_index, to, value, data, 0)

    transaction_count.write(tx_index + 1)

    return ()

end
```

Confirming Transactions

Function for an owner to confirm a transaction.

```cairo
@external
func confirm_transaction(tx_id: felt):
    only_owner()
    require_transaction_exists(tx_id)

    let (to, value, data, executed) = transaction_details.read(tx_id)
    assert executed == 0, 'Transaction already executed'

    confirmations.write(tx_id, syscall_ptr.get_caller_address(), 1)
    return ()
end
```

Executing Transactions

Function to execute a confirmed transaction.

```cairo
@external
func execute_transaction(tx_id: felt):
```

```
    require_transaction_exists(tx_id)

    require_is_confirmed(tx_id)



    let (to, value, data, executed) = transaction_details.read(tx_id)

    assert executed == 0, 'Transaction already executed'



    # Logic to execute the transaction, e.g., sending funds or
interacting with a contract



    transaction_details.write(tx_id, to, value, data, 1)

    return ()

end
```

Helper Functions

Include helper functions for common checks and validations.

```cairo
func only_owner():

   let is_owner_flag = is_owner.read(syscall_ptr.get_caller_address())

   assert is_owner_flag == 1, 'Caller not owner'

   return ()
```

```
func require_transaction_exists(tx_id: felt):
    let (to, value, data, executed) = transaction_details.read(tx_id)
    assert to != 0, 'Transaction does not exist'
    return ()

func require_is_confirmed(tx_id: felt):
    let confirm_count = 0
    for owner in owners:
        confirm_count += confirmations.read(tx_id, owner)
    let required = required_confirmations.read()
    assert confirm_count >= required, 'Not enough confirmations'
    return ()
end
```

Event Logging

Define events for logging transaction submission, confirmation, and execution.

```cairo
```

```
@event

func TransactionSubmitted(tx_id: felt, to: felt, value: felt):

end



@event

func TransactionConfirmed(tx_id: felt, owner: felt):

end



@event

func TransactionExecuted(tx_id: felt):

end
```

Considerations

1. **Ownership Management**: Implement functions to add and remove owners and to change the number of required confirmations.

2. **Transaction Handling**: Ensure that the execution logic securely handles transactions, possibly including interacting with other contracts or sending funds.

3. **Security**: Include multi-signature security measures, checks for reentrancy attacks, and validate inputs thoroughly.

4. **Gas Optimization**: Optimize for gas efficiency, as multi-signature operations may be costly.

5. **Testing and Auditing**: Rigorous testing and professional auditing are essential to ensure the contract's security and reliability.

Conclusion

The outlined contract provides a conceptual framework for a multi-signature wallet. In practice, such contracts can become highly complex, especially when integrating with user interfaces or other blockchain protocols. Development should be accompanied by a thorough understanding of security practices and robust testing methodologies.

10. Basic Structure of a DEX Smart Contract

Creating a decentralized exchange (DEX) smart contract involves significant complexity, particularly if it includes features like liquidity pools, order books, or automated market makers (AMMs). Below is a conceptual outline of how a DEX contract could look in Cairo for StarkNet, focusing on the AMM model which is popular in many current DEX implementations.

Contract Setup

```cairo
%lang starknet
```

```
from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables for liquidity pools and tracking user liquidity.

```cairo
@storage_var
func liquidity_pool(token_a: felt, token_b: felt) -> (token_a_amount: felt, token_b_amount: felt):
end

@storage_var
func user_liquidity(user: felt, token_a: felt, token_b: felt) -> (amount: felt):
end
```

Adding Liquidity

Function to add liquidity to a pool, which could also mint liquidity tokens representing pool shares.

```cairo
@external
func add_liquidity(user: felt, token_a: felt, token_b: felt, amount_a: felt, amount_b: felt):
    # Logic to deposit tokens into the pool
    # Update the pool's liquidity balances
    # Mint liquidity tokens (pool shares) for the user
    return ()
end
```

Removing Liquidity

Function to remove liquidity from a pool and return a proportionate amount of each token to the user.

```cairo
@external
func remove_liquidity(user: felt, token_a: felt, token_b: felt, liquidity: felt):
    # Logic to burn user's liquidity tokens
    # Update the pool's liquidity balances
    # Withdraw a proportionate amount of each token from the pool to the user
    return ()
```

```
end
```

Swapping Tokens

Implement the swap functionality using the constant product formula (x * y = k) for AMMs.

```cairo
@external
func swap(user: felt, token_in: felt, token_out: felt, amount_in: felt):
    # Use the AMM formula to calculate amount_out
    # Update the pool's liquidity balances
    # Transfer token_out to the user
    return ()
end
```

6. **Price Calculation**

Function to calculate the price of a token swap.

```cairo
@view
```

```
func calculate_price(token_in: felt, token_out: felt, amount_in: felt) -
> (amount_out: felt):

    # Use the AMM formula to calculate amount_out

    return (amount_out)

end

```
```

Event Logging

Define events for logging liquidity added/removed and tokens swapped.

```
```cairo

@event

func LiquidityAdded(user: felt, token_a: felt, token_b: felt, amount_a:
felt, amount_b: felt):

end



@event

func LiquidityRemoved(user: felt, token_a: felt, token_b: felt,
amount_a: felt, amount_b: felt):

end



@event
```

```
func TokensSwapped(user: felt, token_in: felt, token_out: felt,
amount_in: felt, amount_out: felt):

end

```
```

Considerations

1. **Token Integration**: The DEX contract must interact with token contracts (e.g., ERC-20) for token transfers.

2. **Slippage Control**: Include mechanisms to control slippage during trades, which can be a major concern for users.

3. **Fees**: Implement a fee model to incentivize liquidity providers and cover operational costs.

4. **Security**: Protect against common DEX vulnerabilities like front-running and ensure contract functions are secure.

5. **Governance**: Optionally, integrate governance mechanisms to allow liquidity providers to vote on important parameters or upgrades.

6. **Scalability**: Design the contract to handle a high volume of trades efficiently.

7. **Testing and Auditing**: Conduct thorough testing and professional auditing to ensure the contract's security and performance.

Testing and Deployment
```

The development of a DEX smart contract requires rigorous testing due to the financial implications and potential for user funds to be at risk. Deploy the contract to a testnet for extensive testing before considering a mainnet launch.

Conclusion

The outline provided offers a conceptual framework for a DEX smart contract, focusing on the AMM model. Developing a real-world DEX requires careful attention to detail, an in-depth understanding of economic models, and a strong emphasis on security. The contract should be designed to be upgradable and maintainable, ensuring long-term sustainability and compliance with evolving blockchain standards.

11. Basic Structure of a Flash Loan Smart Contract

Implementing a flash loan contract in Cairo for StarkNet would involve creating a contract that allows users to borrow assets as long as they are returned within the same transaction, along with any required fees. Here's a simplified outline of a flash loan contract structure:

Contract Setup

```
```cairo

%lang starknet



from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables
```

Define storage variables for keeping track of available funds and fees.

```cairo
@storage_var
func available_funds(token: felt) -> (amount: felt):
end


@storage_var
func flash_loan_fee(token: felt) -> (fee_percentage: felt):
end
```

Flash Loan Function

Function to execute a flash loan, which involves lending out funds and expecting them back with a fee in the same transaction.

```cairo
@external
func flash_loan(token: felt, amount: felt, user_callback: felt, data:
felt):
    let (funds) = available_funds.read(token)
    assert funds >= amount, 'Insufficient funds for flash loan'
```

```
    let (fee_percentage) = flash_loan_fee.read(token)

    let fee = amount * fee_percentage / 100  # Calculate fee based on percentage



    # Logic to lend the amount to the user

    # The user must implement the callback function to use the loaned funds and return them with the fee



    # Verify that the funds and fee are returned

    # Logic to check the returned amount with fee

    # If the check passes, complete the transaction; otherwise, revert



    return ()

end
```

Setting Fees

Function to set the fee for taking out a flash loan.

```cairo
@external

func set_flash_loan_fee(token: felt, fee_percentage: felt):
```

```
    # Only callable by the contract owner or governance mechanism

    flash_loan_fee.write(token, fee_percentage)

    return ()

end

```

Funds Management

Functions for depositing and withdrawing funds from the flash loan provider.

```cairo
@external

func deposit_funds(token: felt, amount: felt):

    # Logic for depositing funds

    let (current_funds) = available_funds.read(token)

    available_funds.write(token, current_funds + amount)

    return ()


@external

func withdraw_funds(token: felt, amount: felt):

    # Only callable by the contract owner or designated role

    let (current_funds) = available_funds.read(token)

    assert current_funds >= amount, 'Insufficient funds to withdraw'
```

```
    available_funds.write(token, current_funds - amount)

    return ()

end
```

Event Logging

Define events for logging flash loan operations, deposits, and withdrawals.

```cairo
@event

func FlashLoanTaken(token: felt, amount: felt, fee: felt):

end



@event

func FundsDeposited(token: felt, amount: felt):

end



@event

func FundsWithdrawn(token: felt, amount: felt):

end
```

Considerations

1. **Security**: Flash loan contracts can be vulnerable to attacks; hence, they need to be thoroughly tested and audited.

2. **Callback Function:** Implement a secure way to call a user-provided callback function that uses the loaned amount and ensures it returns the amount plus the fee.

3. **Reentrancy Guard**: Protect against reentrancy attacks by ensuring that no external calls can interfere with the loan repayment.

4. **Token Integration**: Integrate with token contracts (e.g., ERC-20) for managing the lending and repayment of funds.

5. **Governance**: Set up a governance mechanism for managing parameters such as available funds and loan fees.

Testing and Deployment

Flash loan contracts are high-stakes and complex, and they must be rigorously tested and audited before deployment. Deploy the contract to a testnet and simulate various flash loan use cases and potential attack vectors before going live.

Conclusion

The outlined contract provides a foundational framework for a flash loan contract. It is crucial to approach the development and deployment of such contracts with caution due to their potential risks and complexities.

12. Basic Structure of a Yield Farming Smart Contract

Creating a yield farming contract involves tracking users' deposits into different DeFi protocols and rewarding them accordingly. Below is a high-level outline of how such a contract might look in Cairo, used on StarkNet.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables to keep track of user deposits and reward calculations.

```cairo
@storage_var
func user_deposits(user: felt, protocol: felt) -> (amount: felt):
end
```

```
@storage_var

func reward_rate(protocol: felt) -> (rate: felt):

end



@storage_var

func user_rewards(user: felt) -> (reward: felt):

end
```

Deposit Function

Function for users to deposit into a DeFi protocol.

```cairo
@external

func deposit(user: felt, protocol: felt, amount: felt):

    # You would typically also call the deposit function of the DeFi protocol's smart contract

    let (current_deposit) = user_deposits.read(user, protocol)

    user_deposits.write(user, protocol, current_deposit + amount)

    return ()

end
```

Withdraw Function

Function for users to withdraw their deposits from a DeFi protocol.

```cairo
@external
func withdraw(user: felt, protocol: felt, amount: felt):
    let (current_deposit) = user_deposits.read(user, protocol)
    assert current_deposit >= amount, 'Insufficient balance'
    # You would typically also call the withdraw function of the DeFi protocol's smart contract
    user_deposits.write(user, protocol, current_deposit - amount)
    return ()
end
```

Calculate Rewards

Function to calculate rewards based on deposits and the reward rate.

```cairo
func calculate_rewards(user: felt, protocol: felt):
    let (deposit_amount) = user_deposits.read(user, protocol)
    let (rate) = reward_rate.read(protocol)
    let reward = deposit_amount * rate  # Simplified reward calculation
```

```
    let (current_reward) = user_rewards.read(user)

    user_rewards.write(user, current_reward + reward)

    return ()

end
```

Claim Rewards

Function for users to claim their accumulated rewards.

```cairo
@external

func claim_rewards(user: felt):

    let (reward) = user_rewards.read(user)

    assert reward > 0, 'No rewards to claim'

    # Transfer the rewards to the user

    user_rewards.write(user, 0)

    return ()

end
```

Set Reward Rate

Function to set the reward rate for a DeFi protocol.

```cairo
@external
func set_reward_rate(protocol: felt, rate: felt):
    # Only callable by the contract owner or through a governance mechanism
    reward_rate.write(protocol, rate)
    return ()
end
```

Event Logging

Define events for logging deposits, withdrawals, and reward claims.

```cairo
@event
func DepositEvent(user: felt, protocol: felt, amount: felt):
end

@event
func WithdrawalEvent(user: felt, protocol: felt, amount: felt):
```

```
end



@event

func RewardClaimed(user: felt, reward: felt):

end

```

Considerations

1. **Protocol Integration**: The contract must interact with other DeFi protocols for actual deposit and withdrawal.

2. **Reward Distribution:** Implement a reliable mechanism for calculating and distributing rewards, which could include periodic updates or triggers based on certain conditions.

3. **Security**: Protect against potential vulnerabilities specific to DeFi contracts, like flash loan attacks or reentrancy.

4. **Reward Pool:** Manage a pool of tokens that will be used to distribute rewards to users.

5. **Governance**: Optionally, incorporate governance features to manage the parameters of the yield farming contract.

6. **Testing and Auditing**: Rigorously test the contract for correctness and security, and conduct an audit before deploying to mainnet.

Testing and Deployment

Testing this contract should include simulation of user interactions across various scenarios, including edge cases. This ensures that the rewards are calculated accurately and the contract interacts securely with other protocols.

Conclusion

This outline provides a foundational framework for a yield farming smart contract. In practice, additional features and safeguards would be needed to ensure it operates efficiently and securely in the DeFi ecosystem.

13. Basic Structure of an Escrow Service Smart Contract

An escrow service smart contract acts as a neutral third-party between two or more parties in a transaction. Funds are held by the contract until predefined conditions are met. Below is a simplified conceptual outline of an escrow contract in Cairo for StarkNet.

Contract Setup

```
```cairo

%lang starknet



from starkware.cairo.common.cairo_builtins import HashBuiltin
```
```

Storage Variables

Define storage variables for escrowed funds and the conditions for release.

```cairo
@storage_var
func escrow_balance(transaction_id: felt) -> (amount: felt):
end

@storage_var
func escrow_condition(transaction_id: felt) -> (condition_met: felt):
end
```

Escrow Deposit Function

Function for a party to deposit funds into escrow.

```cairo
@external
func deposit_into_escrow(transaction_id: felt, amount: felt):
    # Logic to transfer funds from the sender to the contract
    escrow_balance.write(transaction_id, amount)
    return ()
```

```
end
```

Set Escrow Conditions

Function to define the conditions under which the funds can be released.

```cairo
@external
func set_escrow_condition(transaction_id: felt, condition: felt):
    # This could be set by the contract creator or through a multi-party consensus mechanism
    escrow_condition.write(transaction_id, condition)
    return ()
end
```

Release Funds

Function to release funds from escrow when conditions are met.

```cairo
@external
func release_funds(transaction_id: felt, to: felt):
```

```
    let (condition_met) = escrow_condition.read(transaction_id)
    assert condition_met == 1, 'Escrow conditions not met'

    let (amount) = escrow_balance.read(transaction_id)
    # Logic to transfer funds from the contract to the recipient
    escrow_balance.write(transaction_id, 0)
    return ()
end
```

Refund

Function to refund the funds to the depositor if the conditions are not met or the transaction is canceled.

```cairo
@external
func refund(transaction_id: felt, to: felt):
    let (condition_met) = escrow_condition.read(transaction_id)
    assert condition_met == 0, 'Escrow conditions have been met, cannot refund'

    let (amount) = escrow_balance.read(transaction_id)
```

```
    # Logic to transfer funds from the contract back to the depositor
    escrow_balance.write(transaction_id, 0)
    return ()
end
```

Event Logging

Define events for logging escrow activities.

```cairo
@event
func EscrowDeposited(transaction_id: felt, amount: felt):
end

@event
func EscrowConditionSet(transaction_id: felt, condition: felt):
end

@event
func FundsReleased(transaction_id: felt, to: felt, amount: felt):
end
```

```
@event

func FundsRefunded(transaction_id: felt, to: felt, amount: felt):

end

```

Considerations

1. **Multiple Parties:** The contract should handle cases with multiple parties involved in the escrow agreement.

2. **Dispute Resolution:** Implement mechanisms for dispute resolution, possibly including arbitration or voting by multiple parties.

3. **Condition Verification:** Develop a robust method for verifying that conditions have been met, possibly including oracles or multi-signature approval.

4. **Security**: Include security checks to prevent unauthorized access to funds or conditions.

5. **Token Handling**: Integrate with token contracts (e.g., ERC-20) if the escrow is for cryptocurrency other than the native one.

6. **Flexibility**: Ensure that conditions can be set flexibly to accommodate various types of transactions.

7. **Testing and Auditing**: Conduct extensive testing and professional auditing due to the contract's financial implications.

Testing and Deployment

Escrow contracts should be tested in various scenarios to ensure that funds are released correctly and conditions are verified accurately. After thorough testing on a testnet, the contract can be deployed to the mainnet.

Conclusion

This outline provides a basic structure for an escrow service smart contract. Real-world implementations would require additional functionality and robustness to handle a variety of escrow arrangements securely and effectively.

14. Basic Structure of a Lottery System Smart Contract

Designing a lottery system as a smart contract requires mechanisms for ticket purchasing, random winner selection, and prize distribution. Below is a high-level outline of a lottery contract in Cairo for StarkNet.

Contract Setup

```
```cairo

%lang starknet
```
```

```
from starkware.cairo.common.cairo_builtins import HashBuiltin

```

Storage Variables

Define storage variables for managing lottery tickets, participants, and the prize pool.

```cairo
@storage_var

func lottery_active() -> (is_active: felt):

end



@storage_var

func lottery_tickets() -> (tickets: felt):

end



@storage_var

func user_tickets(user: felt) -> (ticket_count: felt):

end



@storage_var

func prize_pool() -> (amount: felt):

end
```

```

## Start Lottery

Function to start a new lottery round.

```cairo
@external
func start_lottery():
    # Ensure no active lottery
    let (is_active) = lottery_active.read()
    assert is_active == 0, 'Lottery already active'

    # Set the lottery to active
    lottery_active.write(1)
    # Reset the prize pool and tickets
    prize_pool.write(0)
    lottery_tickets.write(0)
    return ()
end
```
```

Buy Tickets

Function for users to buy lottery tickets.

```cairo
@external
func buy_ticket(user: felt, ticket_amount: felt):
    # Ensure the lottery is active
    let (is_active) = lottery_active.read()
    assert is_active == 1, 'Lottery not active'

    # Logic to transfer funds from the user to the contract as a ticket purchase
    let (current_tickets) = lottery_tickets.read()
    lottery_tickets.write(current_tickets + ticket_amount)

    let (current_user_tickets) = user_tickets.read(user)
    user_tickets.write(user, current_user_tickets + ticket_amount)

    # Increase prize pool
    let (current_prize_pool) = prize_pool.read()
    prize_pool.write(current_prize_pool + TICKET_PRICE * ticket_amount)
    return ()
```

```
end
```

End Lottery and Pick Winner

Function to end the lottery and pick a random winner.

```cairo
@external
func end_lottery():
    # Ensure the lottery is active
    let (is_active) = lottery_active.read()
    assert is_active == 1, 'Lottery not active'

    # Random selection logic to choose a winner
    let winner = select_random_winner()

    # Transfer the prize pool to the winner
    let (total_prize) = prize_pool.read()
    # Transfer logic here

    # Set the lottery to inactive
    lottery_active.write(0)
```

```
    return ()

end

```

Helper Functions

Include helper functions such as for random number generation.

```cairo

func select_random_winner() -> (winner: felt):

    # Implement a random selection mechanism

    # NOTE: True randomness is a complex issue on blockchains; this
often involves oracles or commit-reveal schemes

    return (some_generated_random_winner)

end

```

Event Logging

Define events for logging lottery activities, ticket purchases, and winner selection.

```cairo

@event

func LotteryStarted():

end



```

```
@event

func TicketPurchased(user: felt, ticket_amount: felt):

end



@event

func LotteryEnded(winner: felt, prize_amount: felt):

end
```

Considerations

1. **Randomness**: Implementing true randomness in smart contracts is challenging. You might need to use an oracle or a commit-reveal scheme with off-chain components.

2. **Security**: Protect against potential security issues such as manipulation of the random number generator.

3. **Compliance**: Ensure that the contract adheres to local regulations regarding lotteries and gambling.

4. **Fairness**: Ensure that the lottery system is fair and transparent, allowing all participants an equal chance of winning based on their tickets.

5. **Testing and Auditing**: Rigorously test the contract for correctness, especially the random number generation, and conduct an audit before deployment.

Testing and Deployment

The lottery contract should undergo thorough testing on a testnet, including the random winner selection process and the proper transfer of the prize pool. After extensive testing and any necessary audits, the contract can be deployed to the mainnet.

Conclusion

This outline provides a foundational framework for a lottery system smart contract. Real-world implementation would need to address the challenges of randomness and compliance with gambling regulations. It's essential to ensure fairness and security for all participants in the lottery.

15. Basic Structure of a Crowdfunding/ICO Smart Contract

Creating a smart contract for managing a crowdfunding campaign or an Initial Coin Offering (ICO) involves receiving contributions, tracking the funding goal, and distributing tokens to contributors. Below is a high-level outline of a crowdfunding/ICO contract in Cairo for StarkNet.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables to keep track of contributions, goals, and token distribution.

```cairo
@storage_var
func funding_goal() -> (amount: felt):
end


@storage_var
func total_raised() -> (amount: felt):
end


@storage_var
func contribution(user: felt) -> (amount: felt):
```

```
end



@storage_var

func token_distribution(user: felt) -> (amount: felt):

end

```

Initialize ICO

Function to set the funding goal and start the ICO.

```cairo

@external

func initialize_ico(goal_amount: felt):

    # Only callable by the contract owner or initialization mechanism

    funding_goal.write(goal_amount)

    total_raised.write(0)

    return ()

end

```

Contribute to ICO

Function for contributors to send funds to the ICO.

```cairo
@external
func contribute(user: felt, amount: felt):
    # Logic to accept the contribution (e.g., transferring funds to the contract)
    let (current_total) = total_raised.read()
    total_raised.write(current_total + amount)

    let (user_contribution) = contribution.read(user)
    contribution.write(user, user_contribution + amount)
    return ()
end
```

Check Funding Goal

Function to check if the funding goal has been met.

```cairo
@view
func check_funding_goal() -> (goal_met: felt):
```

```
    let (goal) = funding_goal.read()

    let (raised) = total_raised.read()

    return (goal <= raised)  # Returns 1 if goal is met, otherwise 0

end
```

Distribute Tokens

Function to distribute tokens to contributors after the ICO ends.

```cairo
@external

func distribute_tokens():

    # Ensure funding goal is met

    let (goal_met) = check_funding_goal()

    assert goal_met == 1, 'Funding goal not met'



    # Logic to distribute tokens proportionally based on contributions

    # This might involve calling the token contract's mint or transfer function

    return ()

end
```

```
#### 7. **Refund Contributions**

Function to refund contributions if the funding goal is not met.

```cairo

@external

func refund():

    # Ensure funding goal is not met

    let (goal_met) = check_funding_goal()

    assert goal_met == 0, 'Funding goal met, no refunds'



    # Logic to refund contributions

    # This might involve calling a token contract's transfer function to return funds

    return ()

end

```
```

Event Logging

Define events for logging contributions, ICO status, and token distribution.

```
```cairo

@event
```
```

```
func ContributionReceived(user: felt, amount: felt):

end



@event

func IcoEnded(success: felt):

end



@event

func TokensDistributed(user: felt, amount: felt):

end

```

Considerations

1. **Token Handling**: The contract should integrate with a token contract to handle the minting and transferring of ICO tokens.

2. **Funding Goal Logic:** Implement logic to handle what happens when the funding goal is met or not met within the ICO period.

3. **Contribution Caps**: Optionally, set individual or total contribution caps.

4. **Security**: Include security measures to protect against common vulnerabilities like reentrancy attacks.

5. **Compliance**: Ensure the ICO contract complies with relevant regulations and legal requirements.

6. **Testing and Auditing**: Conduct rigorous testing, especially for the token distribution logic and edge cases like reaching the funding goal at the last moment.

Testing and Deployment

Test the contract thoroughly on a testnet, simulating various funding scenarios to ensure that contributions are handled correctly and tokens are distributed as intended. After successful testing and potential audits, deploy the contract to the mainnet.

Conclusion

The outline provided serves as a foundational framework for a crowdfunding/ICO smart contract. The actual implementation would need to address numerous additional details and legal considerations, particularly regarding token economics and regulatory compliance.

16. Basic Structure of a Subscription Service Smart Contract

A subscription service contract is designed to handle periodic payments for ongoing services. It requires functions to initiate, maintain, and cancel subscriptions, as well as to process payments according to the subscription terms. Below is a conceptual outline of such a contract in Cairo for StarkNet.

Contract Setup

```cairo
%lang starknet


from starkware.cairo.common.cairo_builtins import HashBuiltin
```

Storage Variables

Define storage variables to keep track of subscriber information, subscription plans, and payments.

```cairo
@storage_var
```

```
func subscription_plans(plan_id: felt) -> (price: felt, frequency:
felt):

end



@storage_var

func subscriber_info(user: felt) -> (plan_id: felt, next_payment_date:
felt, active: felt):

end



@storage_var

func payment_history(user: felt, payment_id: felt) -> (amount: felt,
date: felt):

end

```

Define Subscription Plans

Function to create new subscription plans with a price and payment frequency.

```
```cairo

@external

func create_subscription_plan(plan_id: felt, price: felt, frequency:
felt):

    # Only callable by the contract owner or through a governance
mechanism

```
```

```
    subscription_plans.write(plan_id, price, frequency)

    return ()

end
```

Initiate Subscription

Function for users to initiate a subscription.

```cairo
@external

func initiate_subscription(user: felt, plan_id: felt):

    let (price, frequency) = subscription_plans.read(plan_id)

    assert price != 0 and frequency != 0, 'Invalid subscription plan'



    # Logic to transfer the initial payment from the user to the
contract

    subscriber_info.write(user, plan_id, get_current_time() + frequency,
1)

    return ()

end
```

Process Subscription Payment

Function to process recurring subscription payments.

```cairo
@external

func process_subscription_payment(user: felt):

    let (plan_id, next_payment_date, active) =
subscriber_info.read(user)

    assert active == 1, 'Subscription not active'

    assert get_current_time() >= next_payment_date, 'Payment not due
yet'



    let (price, frequency) = subscription_plans.read(plan_id)



    # Logic to transfer the subscription fee from the user to the
contract

    # Update the next payment date

    subscriber_info.write(user, plan_id, get_current_time() + frequency,
active)



    # Record the payment

    let payment_id = get_next_payment_id(user)

    payment_history.write(user, payment_id, price, get_current_time())
```

```
    return ()

end

```

Cancel Subscription

Function for users to cancel their subscription.

```cairo
@external

func cancel_subscription(user: felt):

    let (plan_id, next_payment_date, active) =
subscriber_info.read(user)

    assert active == 1, 'Subscription not active'



    subscriber_info.write(user, plan_id, next_payment_date, 0)

    return ()

end

```

Helper Functions

Include helper functions for time handling and payment ID generation.

```cairo
func get_current_time() -> (current_time: felt):
    # Implement logic to get the current time, possibly using a time oracle
    return (current_time_placeholder)

func get_next_payment_id(user: felt) -> (payment_id: felt):
    # Implement logic to generate the next payment ID for a user
    return (payment_id_placeholder)
end
```

Event Logging

Define events for logging subscription activities and payments.

```cairo
@event
func SubscriptionInitiated(user: felt, plan_id: felt):
end

@event
func SubscriptionPaymentProcessed(user: felt, amount: felt):
```

```
end



@event

func SubscriptionCancelled(user: felt):

end

```

Considerations

1. **Payment Handling**: Integrate with a payment processing mechanism or token contract to handle the transfer of subscription fees.

2. **Time-Dependent Logic:** Incorporate a reliable way to handle time within the contract, which might require time oracles.

3. **Plan Management**: Implement functions to manage subscription plan changes.

4. **Security**: Protect against vulnerabilities, especially in payment processing logic.

5. **Compliance**: Ensure the contract complies with regulations concerning subscriptions and periodic payments.

6. **Testing and Auditing**: Conduct thorough testing, especially for time-based logic and payment processing, and consider professional auditing before deployment.

Testing and Deployment

Test the contract extensively on a testnet, verifying that subscriptions can be initiated, payments processed, and subscriptions canceled correctly. After thorough testing and any required audits, the contract can be deployed to the mainnet.

Conclusion

This outline provides a foundational structure for a subscription service smart contract. Real-world implementations would require a more detailed approach,