

Experiment No. 5

Aim: Deploying a Voting/Ballot Smart Contract

Theory:

1. Relevance of require Statements in Solidity Programs

In Solidity, the require statement acts as a guard condition within functions. It ensures that only valid inputs or authorized users can execute certain parts of the code. If the condition inside require is not satisfied, the function execution stops immediately, and all state changes made during that transaction are reverted to their original state. This rollback mechanism ensures that invalid transactions do not corrupt the blockchain data.

For example, in a Voting Smart Contract, require can be used to check:

- Whether the person calling the function has the right to vote (`require(voters[msg.sender].weight > 0, "Has no right to vote");`).
- Whether a voter has already voted before allowing them to vote again.
- Whether the function caller is the chairperson before granting voting rights.

Thus, require statements enforce security, correctness, and reliability in smart contracts. They also allow developers to attach error messages, making debugging and contract interaction easier for users.

2. Keywords: mapping, storage, and memory

- **mapping:**

A mapping is a special data structure in Solidity that links keys to values, similar to a hash table. Its syntax is `mapping(keyType => valueType)`. For example:

```
mapping(address => Voter) public voters;
```

Here, each address (Ethereum account) is mapped to a Voter structure. Mappings are very useful for contracts like Ballot, where you need to associate voters with their data (whether they voted, which proposal they chose, etc.). Unlike arrays, mappings do not have a length property and cannot be iterated over directly, making them gas efficient for lookups but limited for enumeration.

- **storage:**

In Solidity, storage refers to the permanent memory of the contract, stored on the Ethereum blockchain. Variables declared at the contract level are stored in storage by default. Data stored in storage is persistent across transactions, which means once written, it remains available unless

explicitly modified. However, because writing to blockchain storage consumes gas, it is more expensive. For example, a voter's information saved in the voters mapping remains available throughout the contract's lifecycle.

- memory:

In contrast, memory is temporary storage, used only for the lifetime of a function call. When the function execution ends, the data stored in memory is discarded. Memory is mainly used for temporary variables, function arguments, or computations that don't need to be permanently stored on the blockchain. It is cheaper than storage in terms of gas cost. For instance, when handling proposal names or temporary string manipulations, memory is often used.

Thus, a smart contract developer must balance between storage and memory to ensure efficiency and cost-effectiveness.

3. Why bytes32 instead of string?

In earlier implementations of the Ballot contract, bytes32 was used for proposal names instead of string. The reason lies in efficiency and gas optimization.

- bytes32 is a fixed-size type, meaning it always stores exactly 32 bytes of data. This makes storage simple, comparison operations faster, and gas costs lower. However, it limits proposal names to 32 characters, which is not very flexible for user-friendly names.
- string is a dynamically sized type, meaning it can store text of variable length. While it is easier for developers and users (since names can be written normally), it requires more complex handling inside the Ethereum Virtual Machine (EVM). This increases gas usage and may slow down comparisons or manipulations.

To make the system more user-friendly, modern implementations of the Ballot contract often convert from bytes32 to string. Tools like the Web3 Type Converter help developers easily switch between these two types for deployment and testing.

In summary, bytes32 is used when performance and gas efficiency are priorities, while string is preferred for readability and ease of use.

Implementation

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

<**
 * @title Ballot
 * @dev Implements voting process along with vote delegation
 */
contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    struct Proposal {
        string name;
        uint voteCount;
    }

    address public chairperson;

    mapping(address => Voter) public voters;

    Proposal[] public proposals;

    <**
     * @dev Create a new ballot
     */
    constructor(string[] memory proposalNames) {
        require(proposalNames.length > 0, "Proposals required");

        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
```

```
        name: proposalNames[i],
        voteCount: 0
    })));
}
}

/**
 * @dev Give right to vote
 */
function giveRightToVote(address voter) external {
    require(msg.sender == chairperson, "Only chairperson allowed");
    require(voter != address(0), "Invalid address");
    require(!voters[voter].voted, "Already voted");
    require(voters[voter].weight == 0, "Already has voting right");

    voters[voter].weight = 1;
}

/**
 * @dev Delegate vote
 */
function delegate(address to) external {
    require(to != address(0), "Invalid address");

    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "No right to vote");
    require(!sender.voted, "Already voted");
    require(to != msg.sender, "Self delegation not allowed");

    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;
        require(to != msg.sender, "Delegation loop found");
    }

    Voter storage delegate_ = voters[to];
    require(delegate_.weight >= 1, "Delegate has no right");

    sender.voted = true;
    sender.delegate = to;
```

```
if (delegate_.voted) {
    proposals[delegate_.vote].voteCount += sender.weight;
}
else {
    delegate_.weight += sender.weight;
}
}

/** 
 * @dev Cast vote
 */
function vote(uint proposal) external {
    require(proposal < proposals.length, "Invalid proposal");

    Voter storage sender = voters[msg.sender];

    require(sender.weight != 0, "No right to vote");
    require(!sender.voted, "Already voted");

    sender.voted = true;
    sender.vote = proposal;

    proposals[proposal].voteCount += sender.weight;
}

/** 
 * @dev Find winning proposal
 */
function winningProposal() public view returns (uint winningProposal_) {
    uint winningVoteCount = 0;

    for (uint p = 0; p < proposals.length; p++) {
        if (proposals[p].voteCount > winningVoteCount) {
            winningVoteCount = proposals[p].voteCount;
            winningProposal_ = p;
        }
    }
}
```

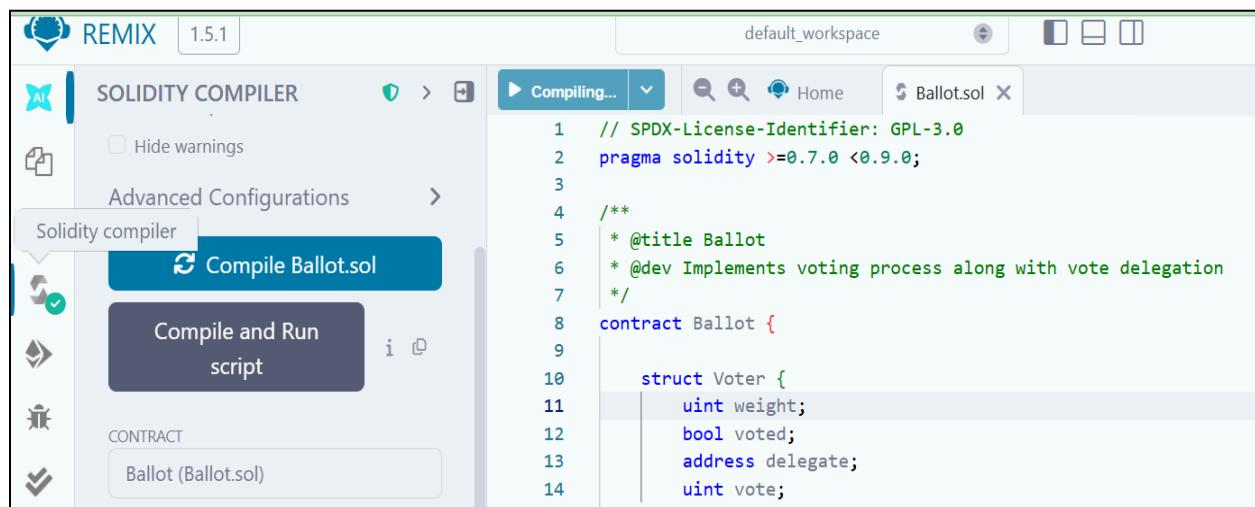
```

/**
 * @dev Returns winner name
 */
function winnerName() external view returns (string memory winnerName_) {
    winnerName_ = proposals[winningProposal()].name;
}

/**
 * @dev Returns total number of proposals
 */
function getProposalCount() public view returns(uint) {
    return proposals.length;
}
}

```

Compiled Ballot.sol Contract



The screenshot shows the REMIX Solidity Compiler interface version 1.5.1. The left sidebar has icons for file operations, a warning icon, advanced configurations, and a voter icon. The main area is titled 'SOLIDITY COMPILER' with a 'Solidity compiler' dropdown set to 'Solidity compiler'. A blue button labeled 'Compile Ballot.sol' is highlighted. Below it is a dark grey button labeled 'Compile and Run script'. The right panel shows the Solidity code for 'Ballot.sol' with line numbers 1 through 14. The code includes SPDX license information, a contract definition for 'Ballot', and a struct definition for 'Voter'.

```

1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.7.0 <0.9.0;
3
4 /**
5  * @title Ballot
6  * @dev Implements voting process along with vote delegation
7  */
8 contract Ballot {
9
10     struct Voter {
11         uint weight;
12         bool voted;
13         address delegate;
14         uint vote;
}

```

Deploying and running of the contract

Loading the Proposal Candidate's Names (string)

The screenshot shows the Remix IDE interface with the "Explain contract" tab selected. On the left, there are sections for "Estimated Gas" (Custom value: 3000000), "Value" (0 Wei), and "Contract" (Ballot - contracts/3_Ballot.sol). Below these are buttons for "Deploy" (with accounts: "Akriti", "Sannidhi", "Shravani") and "At Address". The main area displays the Remix 1.5.1 welcome screen, which includes a terminal-like interface for interacting with the Ethereum blockchain. It shows a transaction record for deploying the Ballot contract with address 0x5B3...eddC4.

Giving the right to an account other than and by the chairman

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/**
```

Explain contract

Your files are stored in indexedDB, 1.16 MB / 205.6 GB used

- Check transactions details and start debugging.
- Execute JavaScript scripts:
 - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.execCurrent()` in the command line interface
 - Right-click on a JavaScript file in the file explorer and then click 'Run'

The following libraries are accessible:

- ethers.js

Type the library name to see available commands.

creation of Ballot pending...

[vm] from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0
hash: 0xd71...b51ac creation of Ballot pending...

[vm] from: 0xAB8...35cb2 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0
hash: 0x0d4...7259c

Selecting the account which was given the right to vote and then writing the proposal candidate's index to vote.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/**
```

Explain contract

- Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.execCurrent()` in the command line interface

The following libraries are accessible:

- ethers.js

Type the library name to see available commands.

creation of Ballot pending...

[vm] from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0
hash: 0xd71...b51ac creation of Ballot pending...

[vm] from: 0xAB8...35cb2 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0
hash: 0x0d4...7259c transact to Ballot.vote pending ...

[vm] from: 0x5B3...eddC4 to: Ballot.vote(uint256) 0x9d7...b5e99 value: 0 wei data: 0x012...00002 logs: 0
hash: 0x3cf...a9643

Already voted , cant vote again!

```

1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6 */

```

Deployed Contracts

- BALLOT AT 0X9D7...B5E99 (MEM)
- Balance: 0 ETH
- delegate address to
- giveRightToVote 0x4B209938c481177ec7E8f...
- vote 2
- chairperson
- proposals uint256
- voters address
- winnerName
- winningProposal

Low level interactions

CALldata

Transact

Explain contract

creation of Ballot pending...

[vm] from: 0xAB8...35cb2 to: Ballot.(constructor) value: 0 wei data: 0x608...0000 logs: 0 hash: 0xd4...7259c

transact to Ballot.vote pending ...

[vm] from: 0x583...eddC4 to: Ballot.vote(uint256) 0x9D7...b5E99 value: 0 wei data: 0x012...00002 logs: 0 hash: 0x3cf...a9643

transact to Ballot.vote pending ...

[vm] from: 0x4B2...C02db to: Ballot.vote(uint256) 0x9D7...b5E99 value: 0 wei data: 0x012...00002 logs: 0 hash: 0x827...7068c

transact to Ballot.vote errored: Error occurred: revert.

revert

The transaction has been reverted to the initial state.
Reason provided by the contract: "No right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

winnerName:

```

1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6 */

```

Deploy & RUN TRANSACTIONS

0: address: 0x5B38Da6a701c568545 dCfcB03FcB875f56beddC4

PROPOSALS

: uint256

chairperson

0: address: 0x4B209938c481177ec7E8f571ceCaE8A9e22C02db

call

voters address

winnerName

0: string: Shravani

winningProposal

0: uint256: winningProposal_2

Low level interactions

CALldata

Transact

BALLOT AT 0XB30...EA098 (MEM)

Explain contract

transact to Ballot.vote errored: Error occurred: revert.

revert

The transaction has been reverted to the initial state.
Reason provided by the contract: "No right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

call to Ballot.winnerName

call [call] from: 0x4B209938c481177ec7E8f571ceCaE8A9e22C02db to: Ballot.winnerName() data: 0xe2b...a53f0

call to Ballot.winningProposal

call [call] from: 0x4B209938c481177ec7E8f571ceCaE8A9e22C02db to: Ballot.winningProposal() data: 0x609...ff1bd

call to Ballot.chairperson

call [call] from: 0x4B209938c481177ec7E8f571ceCaE8A9e22C02db to: Ballot.chairperson() data: 0x2e4...176cf

Conclusion: In this experiment, a Voting/Ballot smart contract was deployed using Solidity on the Remix IDE. The concepts of require statements, mapping, and data location specifiers like storage and memory were explored to understand their role in ensuring security, efficiency, and correctness in smart contracts. The difference between using bytes32 and string for proposal names was also studied, highlighting the trade-off between gas efficiency and readability. Overall, the experiment provided practical insights into the design and deployment of voting contracts on the blockchain.