

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.

Code:

```
// SPDX-License-Identifier: GPL-3.0  
pragma solidity ^0.8.20;
```

```
/**
 * @title Ballot
 * @dev Implements voting process along with vote delegation
 */
//Prajjwal Pandey D20A / 37
contract Ballot {

    struct Voter {
        uint weight;    // weight is accumulated by delegation
        bool voted;     // if true, that person already voted
        address delegate; // person delegated to
        uint vote;      // index of the voted proposal
    }

    struct Proposal {
        string name;    // short name of proposal
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    /**
     * @dev Create a new ballot to choose one of `proposalNames`.
     */
    constructor(string[] memory proposalNames) {
        require(proposalNames.length > 0, "Must provide proposals");

        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 `voter` the right to vote on this ballot.
     * Can only be called by chairperson.
     */
}
```

```
function giveRightToVote(address voter) external {
    require(msg.sender == chairperson, "Only chairperson can give right to vote");
    require(!voters[voter].voted, "The voter already voted");
    require(voters[voter].weight == 0, "Voter already has right to vote");

    voters[voter].weight = 1;
}

/**
 * @dev Delegate your vote to the voter `to`.
 */
function delegate(address to) external {
    Voter storage sender = voters[msg.sender];

    require(sender.weight != 0, "You have no right to vote");
    require(!sender.voted, "You already voted");
    require(to != msg.sender, "Self-delegation is disallowed");

    // Follow delegation chain
    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;
        require(to != msg.sender, "Found loop in delegation");
    }

    Voter storage delegate_ = voters[to];

    require(delegate_.weight >= 1, "Delegate has no right to vote");

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

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

/**
 * @dev Give your vote (including delegated votes) to proposal `proposal`.
 */
function vote(uint proposal) external {
    require(proposal < proposals.length, "Invalid proposal");

    Voter storage sender = voters[msg.sender];
```

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

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

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

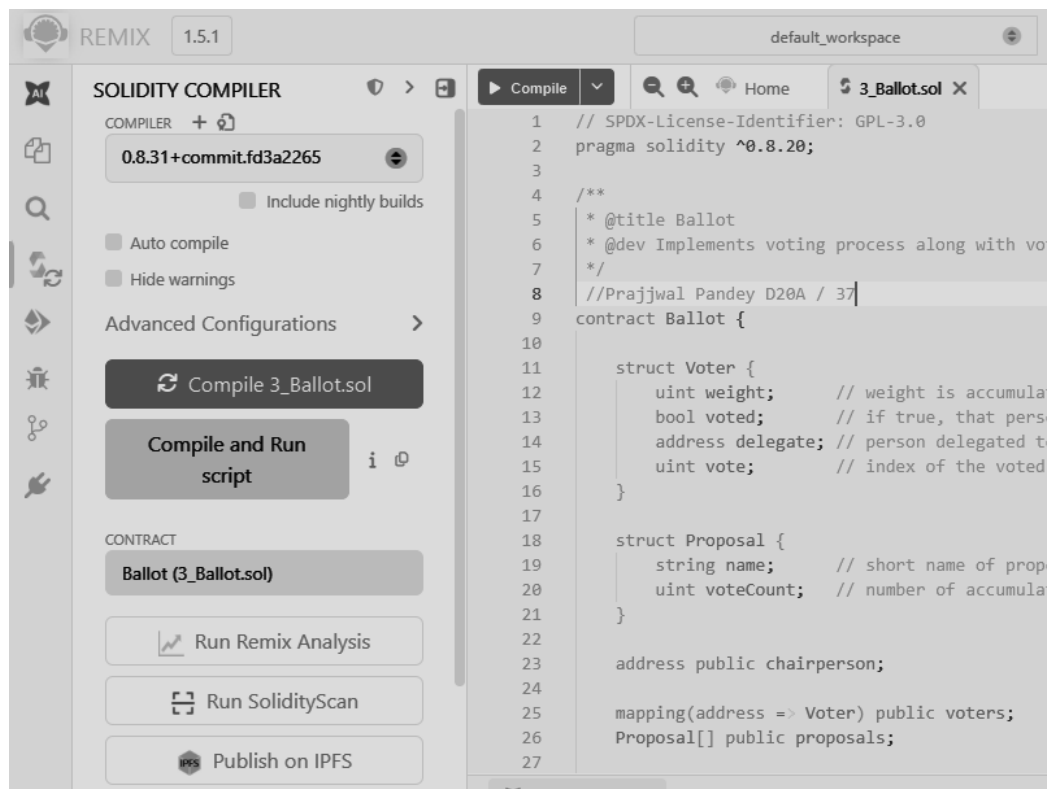
/**
 * @dev Computes the winning proposal.
 */
function winningProposal() public view returns (uint winningProposal_) {
    uint winningVoteCount = 0;

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

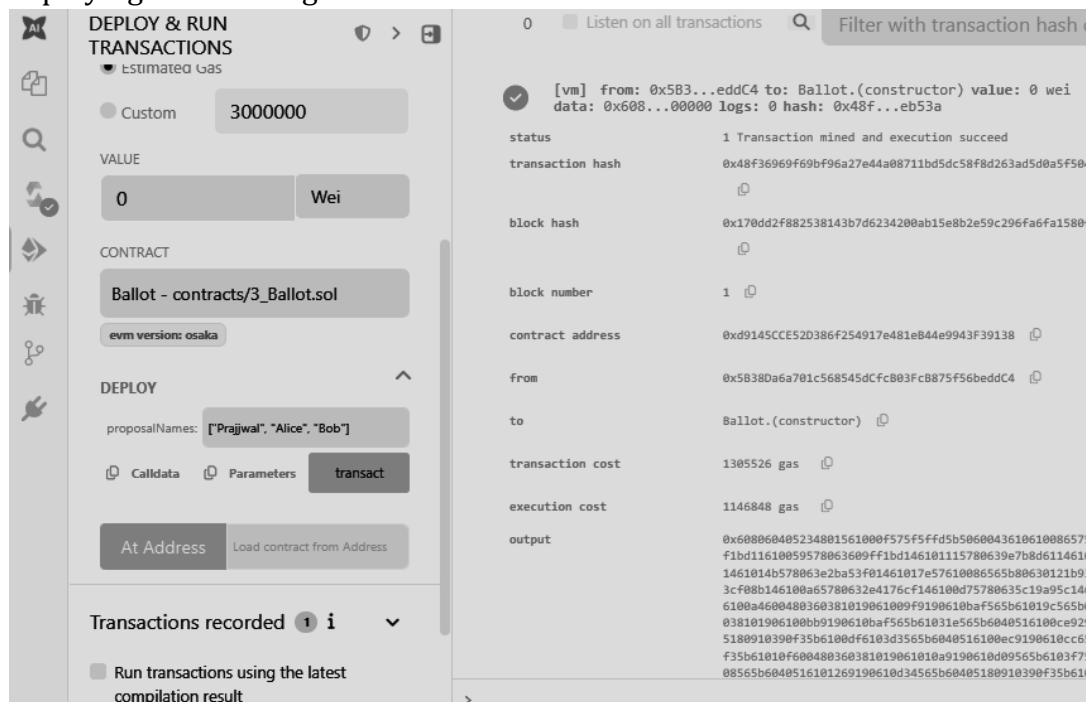
/**
 * @dev Returns the name of the winning proposal.
 */
function winnerName() external view returns (string memory winnerName_) {
    winnerName_ = proposals[winningProposal()].name;
}
}
```

Output Screenshot:

1. Compiled Ballot.sol contract



2. Deploying and running the contract



3. Loading the Proposal Candidate's Names (string)

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN' panel is active, displaying the 'Deployed Contracts' section for a contract named 'BALLOT AT 0XD91...39138 (ME)'. The contract's balance is 0 ETH. The 'vote' function is highlighted, showing a value of 2. Below the 'vote' function, the 'winnerName' function is shown with a value of 'Prajjwal'. The 'winningProposal' function is also shown with a value of 0. The 'PROPOSALS' section is visible, showing a list of proposals with a value of '2'. The 'Call data' and 'Parameters' sections are also visible. On the right, the '3_Ballots.sol' contract code is displayed, showing the 'winningProposal' and 'winnerName' functions. The 'winnerName' function is highlighted, showing its implementation.

```
101 proposals[proposal].voteCount += sender.weight;
102 }
103
104 /**
105  * @dev Computes the winning proposal.
106  */
107 function winningProposal() public view returns (uint winningP
108     uint winningVoteCount = 0;
109
110     for (uint p = 0; p < proposals.length; p++) {
111         if (proposals[p].voteCount > winningVoteCount) {
112             winningVoteCount = proposals[p].voteCount;
113             winningProposal_ = p;
114         }
115     }
116 }
117
118 /**
119  * @dev Returns the name of the winning proposal.
120  */
121 function winnerName() external view returns (string memory wi
```

4. Viewing the details of the Proposal Candidate

The screenshot shows the 'DEPLOY & RUN' panel in the Remix IDE. The 'Deployed Contracts' section is active, showing the 'BALLOT AT 0XD91...39138 (ME)' contract. The 'vote' function is highlighted, showing a value of 2. Below the 'vote' function, the 'winnerName' function is shown with a value of 'Prajjwal'. The 'winningProposal' function is also shown with a value of 0. The 'PROPOSALS' section is visible, showing a list of proposals with a value of '2'. The 'Call data' and 'Parameters' sections are also visible. The 'call' button is highlighted.

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

DEPLOY & RUN
TRANSACTIONS

Deployed Contracts 1

BALLOT AT 0XD91...39138 (ME)

Balance: 0 ETH

delegateaddress to

GIVERIGHTTOVOTE

voter: "Prajjwal"

CalldataParameterstransact

vote2

chairperson

PROPOSALS

: "2"

CalldataParameterscall

votersaddress

winnerName

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

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. Under 'Deployed Contracts', the 'BALLOT AT 0XD91...39138 (ME)' contract is selected. The 'Balance' is 0 ETH. The 'delegate' button is highlighted. Below, the 'GIVERIGHTTOVOTE' contract is expanded, showing a 'voter' field with the address '0xAb8483F64d9C6d1EcF9b849Ae6'. The 'transact' button is highlighted. Below that, the 'vote' button is highlighted with the value '2' in the adjacent field. The 'chairperson' button is also visible. The 'PROPOSALS' section shows a field with the value '2' and a 'call' button. At the bottom, the 'voters' button is highlighted with an 'address' field.

7. We can view the Voter's Information

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface with the transaction details for the 'GIVERIGHTTOVOTE' contract. The 'VOTE' section is expanded, showing a 'proposal' field with the value '0' and a 'transact' button. The 'chairperson' button is also visible. The 'PROPOSALS' section shows a field with the value '2' and a 'call' button. The transaction details on the right include:

Field	Value
transaction hash	0x53130d35c3b4f97b1f77560d71cb9628788cd1e9edb2aec3ab93d2682e8b8575
block hash	0x579700816fb3215f7f4e4a065027dd77ab42f5e6ef3394d6006b3ca480df751c
block number	3
from	0xAb8483F64d9C6d1EcF9b849Ae677d3315835cb2
to	Ballot.vote(uint256) 0xd9145CCE52D386f254917e481e844e9943f39138
transaction cost	73115 gas
execution cost	51923 gas
output	0x
decoded input	{ "uint256 proposal1": "0" }
decoded output	{}
logs	[]
raw logs	[]

8. Selecting the account of the delegator and then writing the address of the voter to be delegated to.

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. On the left, under 'Deployed Contracts', the 'BALLOT AT 0XD91...39138 (ME)' contract is selected. The 'GIVERIGHTTOVOTE' function is being interacted with. The 'voter' field is set to '0x4B20993Bc481177ec7E8f571ceC4'. The 'transact' button is highlighted. On the right, the transaction details are displayed, including block number, from, to, transaction cost, execution cost, output, decoded input, decoded output, logs, and raw logs. The transaction is pending.

Field	Value
block number	3
from	0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to	Ballot.vote(uint256) 0xd9145CCE52D386f254917e481e844e9943F39138
transaction cost	73115 gas
execution cost	51923 gas
output	0x
decoded input	{ "uint256 proposal": "0" }
decoded output	{}
logs	[]
raw logs	[]

transact to Ballot.giveRightToVote pending ...

[vm] from: 0x583...eddC4
to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei
data: 0x9e7...c02db logs: 0 hash: 0xf4d...36e0c

9. Proposal Candidate's Information before giving the Delegate's vote.

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. On the left, under 'Deployed Contracts', the 'BALLOT AT 0XD91...39138 (ME)' contract is selected. The 'GIVERIGHTTOVOTE' function is being interacted with. The 'voter' field is set to '0x4B20993Bc481177ec7E8f571ceC4'. The 'proposal' field is set to '1'. The 'transact' button is highlighted. On the right, the transaction details are displayed, including block number, from, to, transaction cost, execution cost, output, decoded input, decoded output, logs, and raw logs. The transaction is pending.

Field	Value
block number	3
from	0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to	Ballot.vote(uint256) 0xd9145CCE52D386f254917e481e844e9943F39138
transaction cost	73115 gas
execution cost	51923 gas
output	0x
decoded input	{ "uint256 proposal": "0" }
decoded output	{}
logs	[]
raw logs	[]

transact to Ballot.giveRightToVote pending ...

[vm] from: 0x583...eddC4
to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei
data: 0x9e7...c02db logs: 0 hash: 0xf4d...36e0c

transact to Ballot.vote pending ...

[vm] from: 0x482...C02db to: Ballot.vote(uint256) 0xd91...39138
value: 0 wei data: 0x012...00001 logs: 0 hash: 0xfb1...c5cd6

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. On the left, the 'GIVERIGHTTOVOTE' contract is selected with a voter address. The 'VOTE' section shows a proposal of 1 and a 'chairperson' button. The 'PROPOSALS' section shows a proposal of 2 and a 'call' button. The 'voters' and 'winnerName' sections show the current state. On the right, the transaction logs show a failed transaction for 'GIVERIGHTTOVOTE' with the error message: 'The transaction has been reverted to the initial state. Reason provided by the contract: "Already voted". If the transaction failed for not having enough gas, try increasing the gas limit gently.'

10. In this final screenshot we can see that after the delegation's vote the weight of one vote of the one voter who was delegated is the number of people who delegated the voter as their voter (Default weight of any voter is 1).

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. The 'chairperson' button is highlighted. The 'PROPOSALS' section shows a proposal of 2 and a 'call' button. The 'voters' and 'winnerName' sections show the current state. The 'winningProposal' section shows the winning proposal of 0. The 'Low level interactions' section is visible at the bottom.

The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' interface. The 'chairperson' button is highlighted. The 'PROPOSALS' section shows a proposal of 0 and a 'call' button. The 'voters' and 'winnerName' sections show the current state. The 'winningProposal' section shows the winning proposal of 0.

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.

