

## 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 >=0.7.0
//      <0.9.0;

/**
 * @title Ballot
 * @dev Implements voting process along with vote delegation
 */
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single
    // voter. 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
    }
    // This is a type for a single proposal.
    struct Proposal {
        // If you can limit the length to a certain number of bytes
        // always use one of bytes1 to bytes32 because they are much
        // cheaper bytes32 name; // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a 'Voter' struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of 'Proposal' structs.
    Proposal[] public proposals;

    /**
     * @dev Create a new ballot to choose one of 'proposalNames'.
     * @param proposalNames names of proposals
     */
    constructor(bytes32[] memory
```

```

        proposalNames) { chairperson =
        msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // 'Proposal({...})' creates a temporary
            // Proposal object and 'proposals.push(...)'
            // appends it to the end of 'proposals'.
            proposals.push(Proposal({
                name:
                proposalNames[i
                ], voteCount: 0
            })));
        }
    }
    /**
     * @dev Give 'voter' the right to vote on this ballot. May only be called
    by 'chairperson'.
     * @param voter address of voter
     */
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to 'false', execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use 'require' to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went
        wrong. 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 the right to vote.");
    voters[voter].weight = 1;
}

/**
 * @dev Delegate your vote to the voter 'to'.
 * @param to address to which vote is delegated
 */
function delegate(address to) external {
    // assigns reference
    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.");

    // Forward the delegation as long as
    // 'to' also delegated.
    // In general, such loops are very dangerous,
    // because if they run too long, they might
    // need more gas than is available in a block.
    // In this case, the delegation will not be executed,
    // but in other situations, such loops might
    // cause a contract to get "stuck" completely. while
    (voters[to].delegate != address(0)) {
        to = voters[to].delegate;

        // We found a loop in the delegation, not allowed. require(to !=
        msg.sender, "Found loop in delegation.");
    }

    Voter storage delegate_ = voters[to];

    // Voters cannot delegate to accounts that cannot vote. require(delegate_.weight >=
    1);

    // Since 'sender' is a reference, this
    //     modifies     'voters[msg.sender]'.
    sender.voted = true; sender.delegate =
    to;

    if (delegate_.voted) {
        // If the delegate already voted,
        // directly add to the number of votes proposals[delegate_.vote].voteCount +=

```

```

        sender.weight;
    } else {
        // If the delegate did not vote yet,
        // add to her weight. delegate_.weight +=
        sender.weight;
    }
}

/**
 * @dev Give your vote (including votes delegated to you) to proposal
'proposals[proposal].name'.
 * @param proposal index of proposal in the proposals array
 */
function vote(uint proposal) external {
    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;

    // If 'proposal' is out of the range of the array,
    // this will throw automatically and revert all
    // changes.
    proposals[proposal].voteCount += sender.weight;
}

/**
 * @dev Computes the winning proposal taking all previous votes into account.
 * @return winningProposal_ index of winning proposal in the proposals array
 */
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 Calls winningProposal() function to get the index of the winner contained
in the proposals array and then

```

Sonam chhabaidiya

D20A 16

```
* @return winnerName_ the name of the winner
    */
    function winnerName() external view returns
        (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}

/**

* ADDITIONAL FEATURE

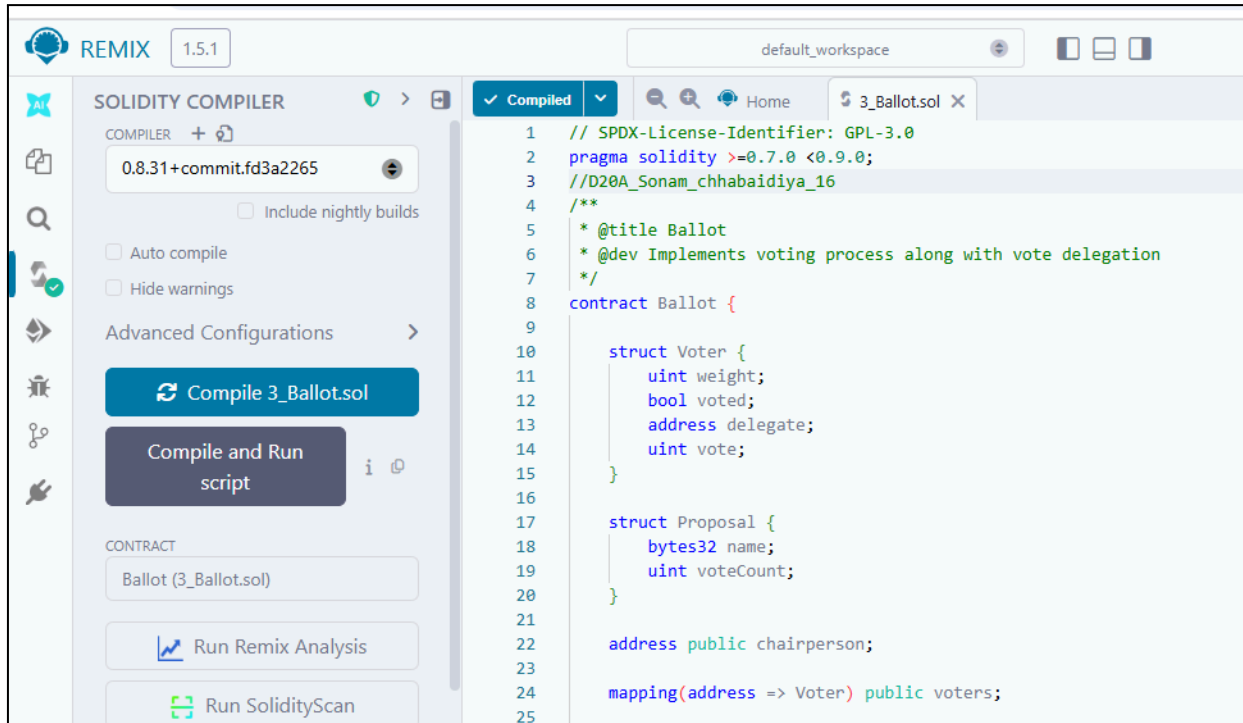
* @dev View all proposal names and vote counts

*/

function getAllProposals() public view returns (string[] memory names, uint256[] memory
voteCounts) {
    names = new string[](proposals.length);
    voteCounts = new uint256[](proposals.length);
    for (uint256 i = 0; i < proposals.length; i++) {
        names[i] = proposals[i].name;
        voteCounts[i] = proposals[i].voteCount;
    }
}
}
```

## Output:

- Compiled Ballot.sol Contract





- Deploying and running of the contract

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active. It shows the contract 'Ballot - contracts/3\_Ballot.sol' with an estimated gas of 3,000,000. The 'Deploy' button is highlighted. Below it, the 'At Address' button is visible. The 'Transactions recorded' section shows 3 transactions, and the 'Deployed Contracts' section shows 1 contract. The main editor displays the Solidity code for '3\_Ballot.sol', which defines a 'Voter' struct, a 'Proposal' struct, and a public 'chairperson' address. The 'Explain contract' panel is open, showing the contract's creation pending. The bottom status bar indicates a successful deployment from 0x5B3...eddC4 to Ballot.(constructor) with a value of 0 wei.

- Viewing the details of the Proposal Candidate using additional function get proposal.

The screenshot shows the Remix IDE interface with the '3\_Ballot.sol' contract loaded. The 'DEPLOY & RUN TRANSACTIONS' panel on the left shows the 'call' button highlighted. The 'GETPROPOSAL' section is active, showing the 'index' set to 0. The 'Calldata' and 'Parameters' tabs are visible, with 'Calldata' showing the string 'name sonam' and 'Parameters' showing 'uint256: voteCount 0'. The 'call' button is highlighted. The main editor displays the Solidity code for '3\_Ballot.sol', showing the 'getProposal' function. The 'Explain contract' panel is open, showing the function's execution. The bottom status bar indicates a successful call from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to Ballot.getProposal with a value of 0x2e4...176cf.

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

The screenshot displays the Remix 1.5.1 IDE interface. On the left sidebar, the 'DEPLOY & RUN TRANSACTIONS' panel is active, showing a deployed contract named 'BALLOT AT 0xE0F...39218 (MEMORY)'. The contract's balance is 0 ETH. Below the contract name, there are buttons for 'delegate', 'giveRightToVote', 'vote', 'chairperson', 'getAllProposals', and 'proposals'. The 'giveRightToVote' button is highlighted with an orange background. The main panel on the right shows the 'Welcome to Remix 1.5.1' message and a list of transactions. The first transaction is a successful call from 0x583...eddC4 to Ballot.constructor, with a value of 0 wei and data 0x608...0000. The second transaction is a successful call from 0x58380a6a701c568545dCfc803Fc8875f56beddC4 to Ballot.chairperson(), with a value of 0 wei and data 0x2e4...176cf. The third transaction is a successful call from 0x583...eddC4 to Ballot.giveRightToVote(address) 0xE0f...39218, with a value of 0 wei and data 0x9e7...35cb2. Each transaction has a 'Debug' button next to it.

REMIX 1.5.1 default\_workspace Login with GitHub Theme

DEPLOY & RUN TRANSACTIONS

At Address Load contract from Address

Transactions recorded 2 i

Deployed Contracts 1

BALLOT AT 0xE0F...39218 (MEMORY)

Balance: 0 ETH

delegate address to

giveRightToVote 0xA8483F64d9C6d1EcF9b849Ae

vote uint256 proposal

chairperson

0: address: 0x58380a6a701c568545dCfc803Fc8875f56beddC4

getAllProposals

proposals uint256

Welcome to Remix 1.5.1

Your files are stored in indexedDB, 1.15 MB / 491.94 MB used

You can use this terminal to:

- Check transactions details and start debugging.
- Execute JavaScript scripts:
  - Input a script directly in the command line interface
  - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.executeCurrent()` 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: 0x583...eddC4 to: Ballot.constructor value: 0 wei data: 0x608...0000 logs: 0 hash: 0xf92...ad2a7 Debug

call to Ballot.chairperson

[call] from: 0x58380a6a701c568545dCfc803Fc8875f56beddC4 to: Ballot.chairperson() data: 0x2e4...176cf Debug

transact to Ballot.giveRightToVote pending ...

[vm] from: 0x583...eddC4 to: Ballot.giveRightToVote(address) 0xE0f...39218 value: 0 wei data: 0x9e7...35cb2 logs: 0 hash: 0x1d0...97ea5 Debug

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

The screenshot shows the REMIX IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active. It contains two main sections: 'GIVERIGHTTOVOTE' and 'VOTE'. The 'VOTE' section has a 'proposal' field with the value '1' and a 'transact' button. Below this, there are fields for 'chairperson' and 'voters'. The right panel shows the 'TRANSACTIONS' list. It displays two transactions. The first transaction is a successful transaction with the following details: [vm] from: 0x5B3...eddC4 to: Ballot.giveRightToVote(address) 0xE0f...39218 value: 0 wei data: 0x9e7...c02d hash: 0xFb2...50434. The second transaction is also successful and shows the following details: [vm] from: 0x4B2...C02db to: Ballot.vote(uint256) 0xE0f...39218 value: 0 wei data: 0x012...00001 logs: 0 hash: 0x3a8...fc382. The transaction details include the status, transaction hash, block hash, block number, from/to addresses, transaction cost, execution cost, output, decoded input, and decoded output.

- We can view the Voter's Information

The screenshot shows the REMIX IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active. It contains two main sections: 'GIVERIGHTTOVOTE' and 'VOTE'. The 'VOTE' section has a 'proposal' field with the value '1' and a 'transact' button. Below this, there are fields for 'chairperson' and 'voters'. The right panel shows the 'TRANSACTIONS' list. It displays two transactions. The first transaction is a successful transaction with the following details: [vm] from: 0x5B3...eddC4 to: Ballot.giveRightToVote(address) 0xE0f...39218 value: 0 wei data: 0x9e7...c02d hash: 0xFb2...50434. The second transaction is also successful and shows the following details: [vm] from: 0x4B2...C02db to: Ballot.vote(uint256) 0xE0f...39218 value: 0 wei data: 0x012...00001 logs: 0 hash: 0x3a8...fc382. The transaction details include the status, transaction hash, block hash, block number, from/to addresses, transaction cost, execution cost, output, decoded input, and decoded output.

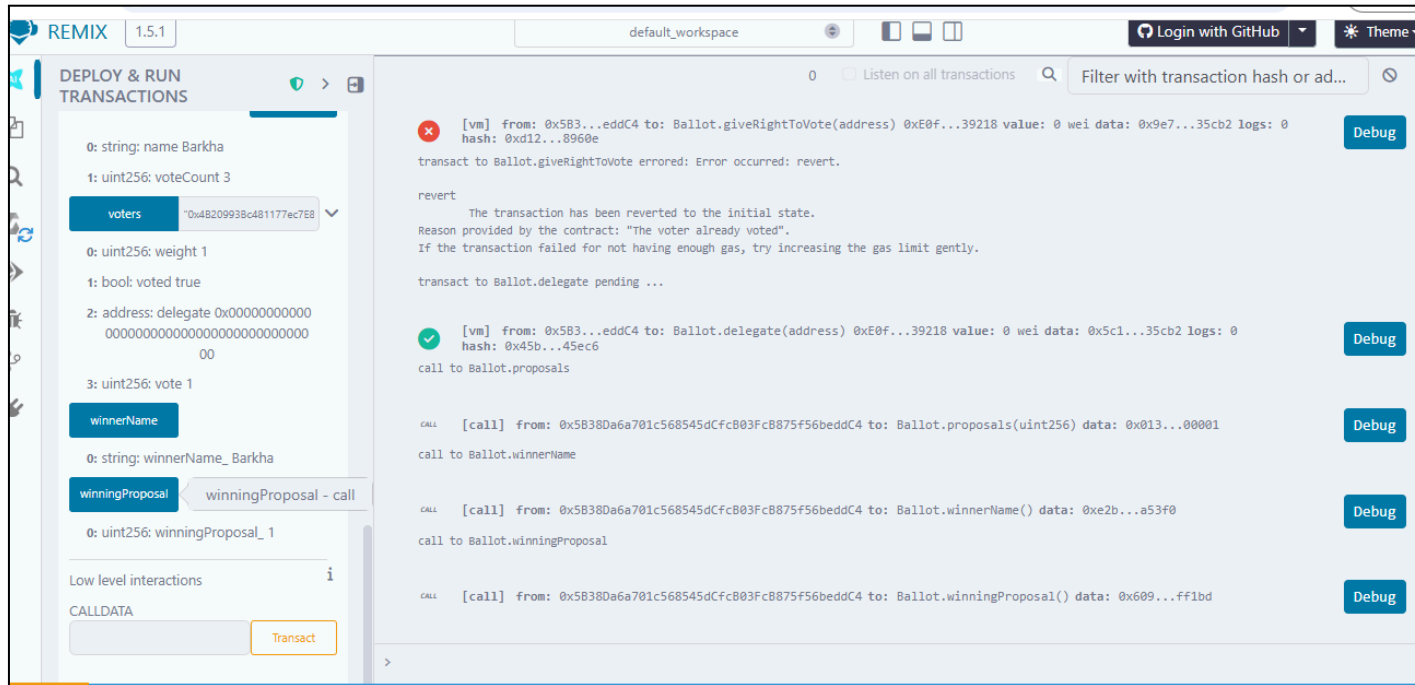
- Selecting the account of the delegator and then writing the address of the voter to be delegated to.

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, displaying the 'DELEGATE' transaction. The 'to' field is set to '0xAb8483F64d9C6d1EcF9b849Ae6'. Below it, the 'GIVERIGHTTOVOTE' transaction is also visible, with the 'voter' field set to the same address. The 'VOTE' transaction is at the bottom. The right sidebar shows the transaction logs. The first log shows a failed transaction to 'Ballot.delegate' with a revert error. The second log shows a failed transaction to 'Ballot.giveRightToVote' with a revert error. The third log shows a successful transaction to 'Ballot.delegate'.

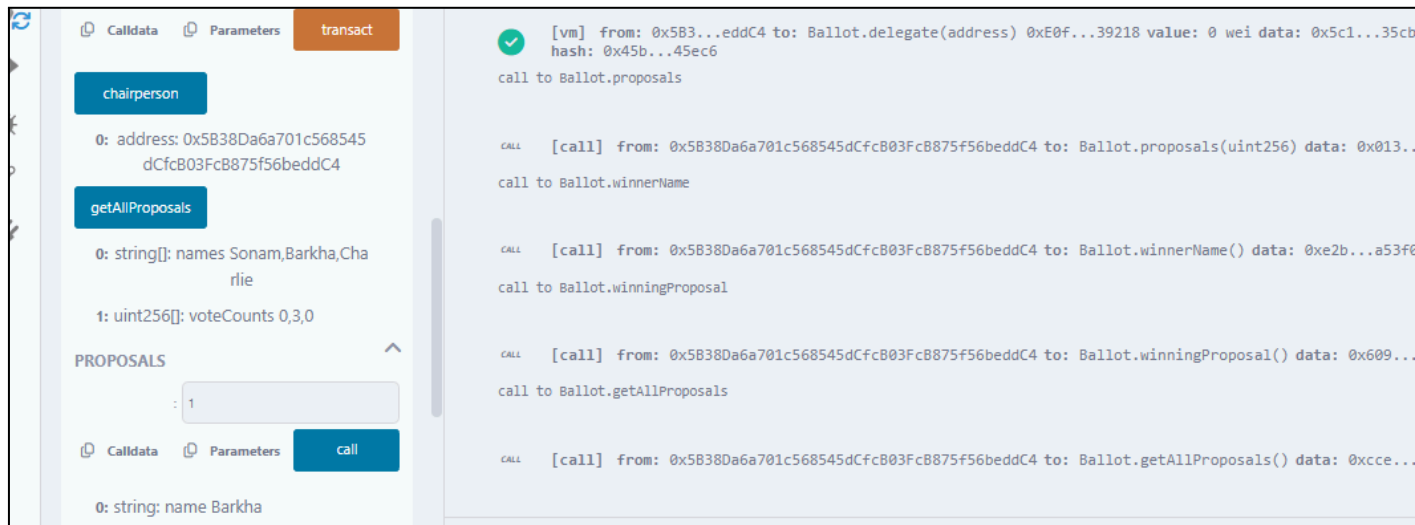
- Proposal Candidate's Information before giving the Delegate's vote

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, displaying the 'PROPOSALS' transaction. The 'proposal' field is set to '1'. Below it, the 'voters' field is set to '0x4B20993Bc481177ec7E8'. The right sidebar shows the transaction logs. The first log shows a call to 'Ballot.winnerName'. The second log shows a call to 'Ballot.winningProposal'. The third log shows a call to 'Ballot.giveRightToVote'.

- 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).



**New additional feature : View all proposal names ,View all vote counts In a single function call.**



**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.