Aim: Deploying a Voting/Ballot Smart Contract

Theory:

1. Relevance of Required 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.

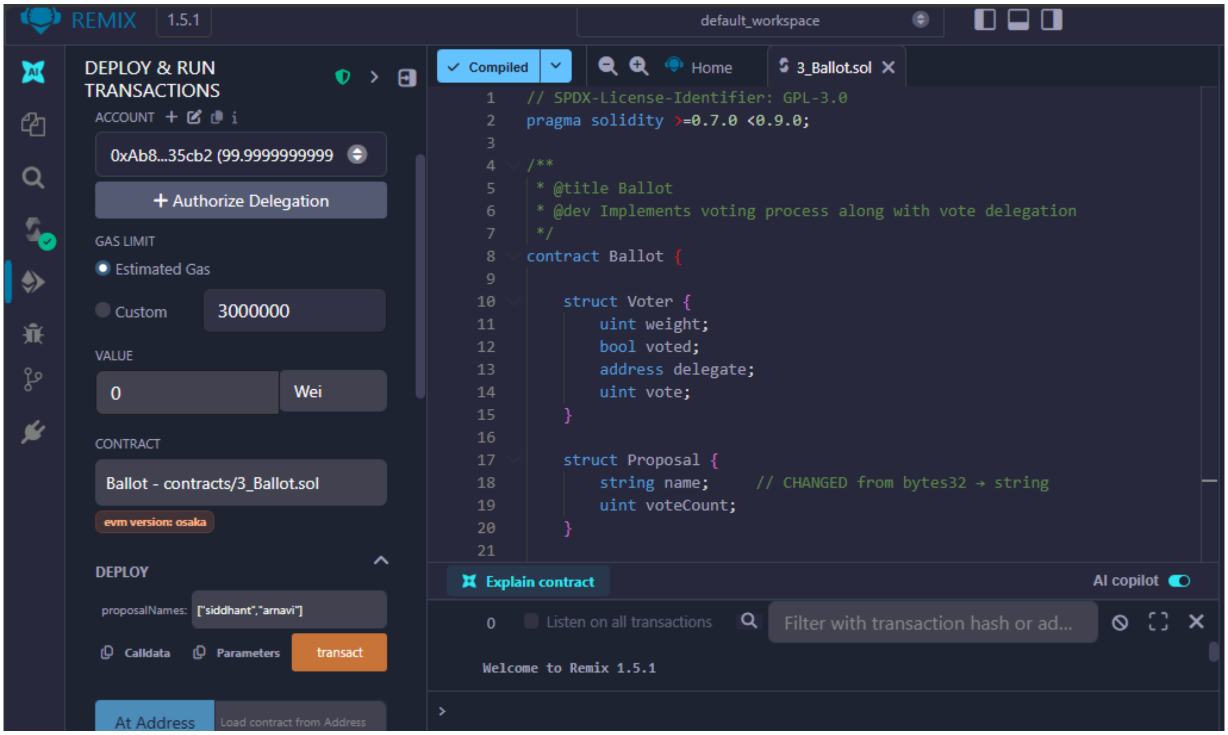
In summary, bytes32 is used when performance and gas efficiency are priorities, while string is preferred for readability and ease of use.

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 {

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

    struct Proposal {
        string name;      // CHANGED from bytes32 → string
        uint voteCount;
    }
    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    /**
     * @dev Create a new ballot
     * @param proposalNames names of proposals
     */
    constructor(string[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(
                Proposal({
                    name: proposalNames[i],
                    voteCount: 0
                })
            );
        }
    }

    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");

        voters[voter].weight = 1;
    }

    function delegate(address to) external {
        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 detected");
        }

        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;
        }
    }

    function vote(uint proposal) external {
        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;
    }

    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;
            }
        }
    }

    function winnerName() external view returns (string memory) {
        return proposals[winningProposal()].name;
    }
}
```

Output:

1. Smart Contract deployed using 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 address.



ACCOUNT: 0xAb8...35cb2 (99.9999999999

GAS LIMIT: ● Estimated Gas ○ Custom 3000000

VALUE: 0 Wei

CONTRACT: Ballot - contracts/3_Ballot.sol

evm version: osaka

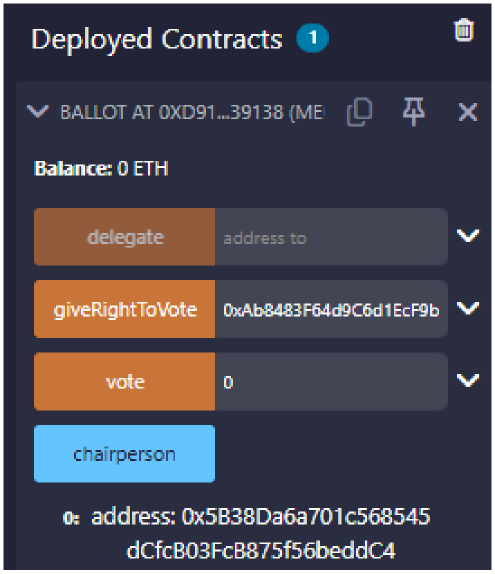
DEPLOY

proposalNames: ["siddhant","arnavi"]

```
// 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;     // CHANGED from bytes32 → string
        uint voteCount;
    }

```

```
0
Welcome to Remix 1.5.1
```

2. Chairperson giving right to vote to another address

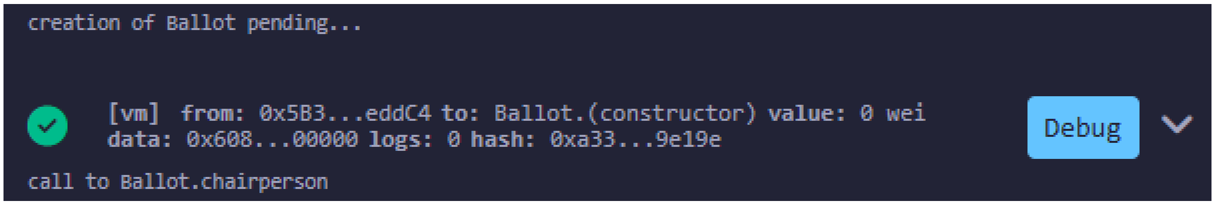


Balance: 0 ETH

giveRightToVote 0xAb8483F64d9C6d1EcF9b

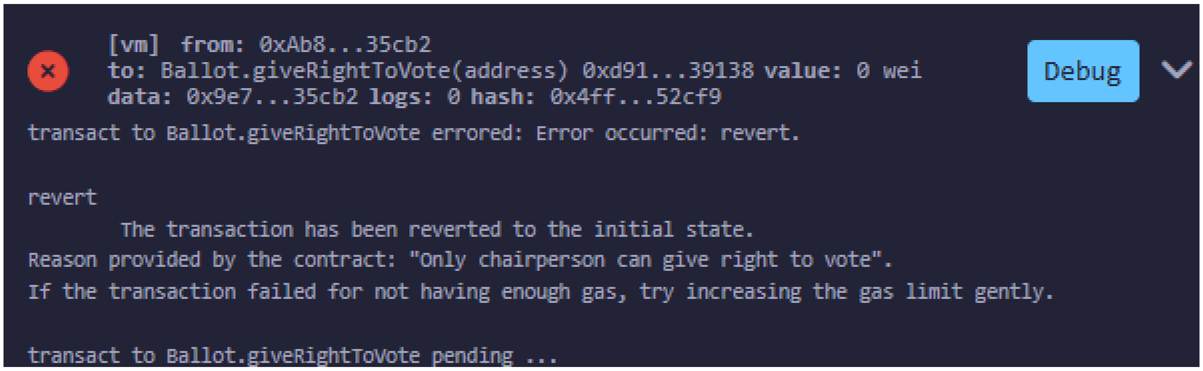
vote 0

0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4



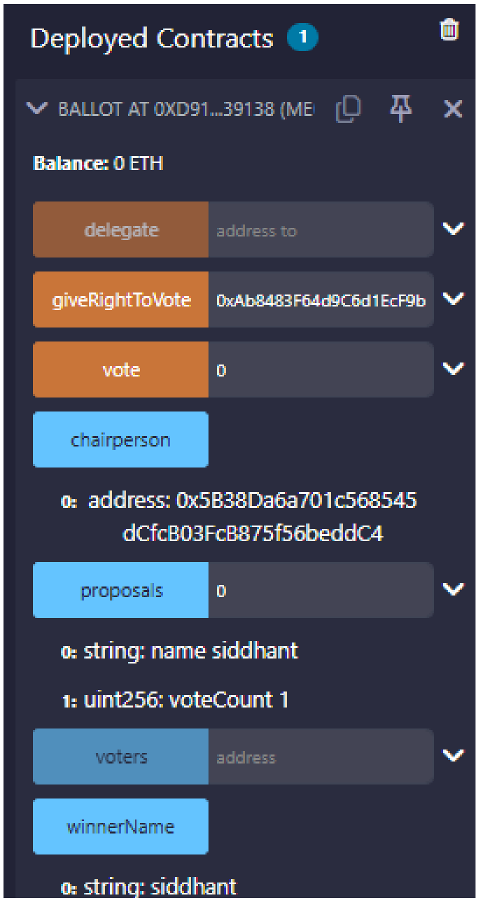
```
creation of Ballot pending...

[vm]  from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei
data: 0x608...00000 logs: 0 hash: 0xa33...9e19e
call to Ballot.chairperson
```

3. Only Chairperson can give voting rights else results in error



```
[vm]  from: 0xAb8...35cb2
to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei
data: 0x9e7...35cb2 logs: 0 hash: 0x4ff...52cf9
transact to Ballot.giveRightToVote errored: Error occurred: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "Only chairperson can give right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

transact to Ballot.giveRightToVote pending ...
```

4. Vote count & results.



Balance: 0 ETH

delegate: address to

giveRightToVote: 0xAb8483F64d9C6d1EcF9b

vote: 0

0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

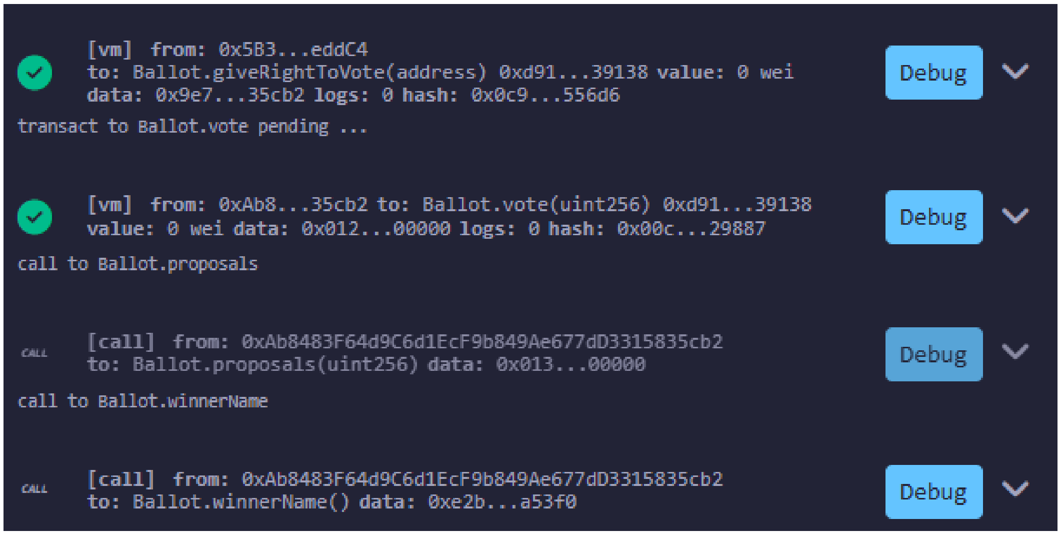
proposals: 0

0: string: name siddhant

1: uint256: voteCount 1

voters: address

0: string: siddhant



```
[vm] from: 0x5B3...eddC4
to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei
data: 0x9e7...35cb2 logs: 0 hash: 0x0c9...556d6
transact to Ballot.vote pending ...

[vm] from: 0xAb8...35cb2 to: Ballot.vote(uint256) 0xd91...39138
value: 0 wei data: 0x012...00000 logs: 0 hash: 0x00c...29887
call to Ballot.proposals

CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to: Ballot.proposals(uint256) data: 0x013...00000
call to Ballot.winnerName

CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to: Ballot.winnerName() data: 0xe2b...a53f0
```

Conclusion: Successfully deployed the contract and used another addresses for casting votes and finalizing results.