

Blockchain Lab

EXP 5

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.

The screenshot shows the Remix Ethereum IDE interface. On the left, there's a sidebar with various icons for deployment, running transactions, and account management. The main area displays the `Ballot.sol` source code:

```

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) {
    infinite gas
    return proposals[winningProposal_].name;
}

```

Below the code editor, there's an "Explain contract" panel with a terminal-like interface. The terminal shows the message: "Welcome to Remix 1.5.1" and "Your files are stored in indexedDB, 1.25 KB / 142.64 GB used".

2. Chairperson giving right to vote to another address

The screenshot shows the Remix Ethereum IDE interface with the GitHub logo at the top. The left sidebar has a "DEPLOY & RUN TRANSACTIONS" section where a transaction is being prepared. The "delegate" button is highlighted in orange. The "GIVERIGHTTOVOTE" section shows fields for "voter" (set to `0xAb8483F64d9C6d1EcF9b849Ae6`), "vote" (set to `1`), and "proposal" (set to `1`). The "Parameters" tab is selected. The "transact" button is orange. The right side shows the transaction details in the terminal-like "Explain contract" panel. The transaction is pending with the following details:

```

logs: 0
hash:
0xd2b...1318e
transact to Ballot.giveRightToVote pending

```

The transaction parameters listed are:

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

A "Debug" button is visible next to the transaction details.

3. Assigned address casting vote for candidate.

The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar has icons for Deploy, Run, Transactions, Contracts, ABI, Events, and Storage. The main area is titled "DEPLOY & RUN TRANSACTIONS". It shows a "voter" field with the value "0xAb8483F64d9C6d1EcF9b849Ae6". Below it are "Calldata" and "Parameters" fields, and a "transact" button. To the right, there's a "VOTE" section with a "proposal" field set to "0", "Calldata" and "Parameters" fields, and a "transact" button. Further down are sections for "chairperson", "proposals" (set to "uint256"), and "VOTERS" (with a field containing "0xAb8483F64d9C6d1EcF9b849Ae6"). On the far right, the code editor shows the Ballot contract source code. The "Compile" button is highlighted. The code includes a for loop to iterate over proposals and an if statement to check if the proposal is voted for. A tooltip "vote - transact (not payable)" points to the "transact" button in the VOTE section. The bottom right pane shows the transaction details: [vm] from: 0xAb8...35cb2 to: Ballot.vote(uint256) 0xd91...39138 value: 0 wei data: 0x012...00000 logs: 0 hash: 0xc66...d2510. A "Debug" button is also present in this pane.

4. Vote count & results.

The screenshot shows the Remix IDE interface with the title bar "REMX 1.5.1". On the left is a sidebar with various icons: AI, Deploy & Run Transactions, Chairperson, Proposals, Voters, Winner Name, and Winning Proposal. The main area displays the "DEPLOY & RUN TRANSACTIONS" section for the "chairperson" role. It shows a proposal with index 0, which has an address of 0x5B38Da6a701c568545 dCfcB03FcB875f56beddC4. Below it, under "proposals", is another entry with index 1, labeled "string: name Diksha" and "uint256: voteCount 1". Under "voters", there are three entries: index 0 with "uint256: weight 1", index 1 with "bool: voted false", and index 2 with "address: delegate 0x00". Under "winnerName", there is one entry with index 0, "string: Diksha". Under "winningProposal", there is one entry with index 0.

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