

EXPERIMENT 5

AIM: Deploying a Voting/Ballot Smart Contract

Tasks:

1. Open [Remix IDE](#)
2. Under **Workspaces**, open **contracts** folder
3. Open **Ballot.sol**, contract.
4. Understand **Ballot.sol** contract.
5. Deploy the contract by changing the Proposal name from **bytes32 → string**

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.

- **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.

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:

Step 1: Open Remix

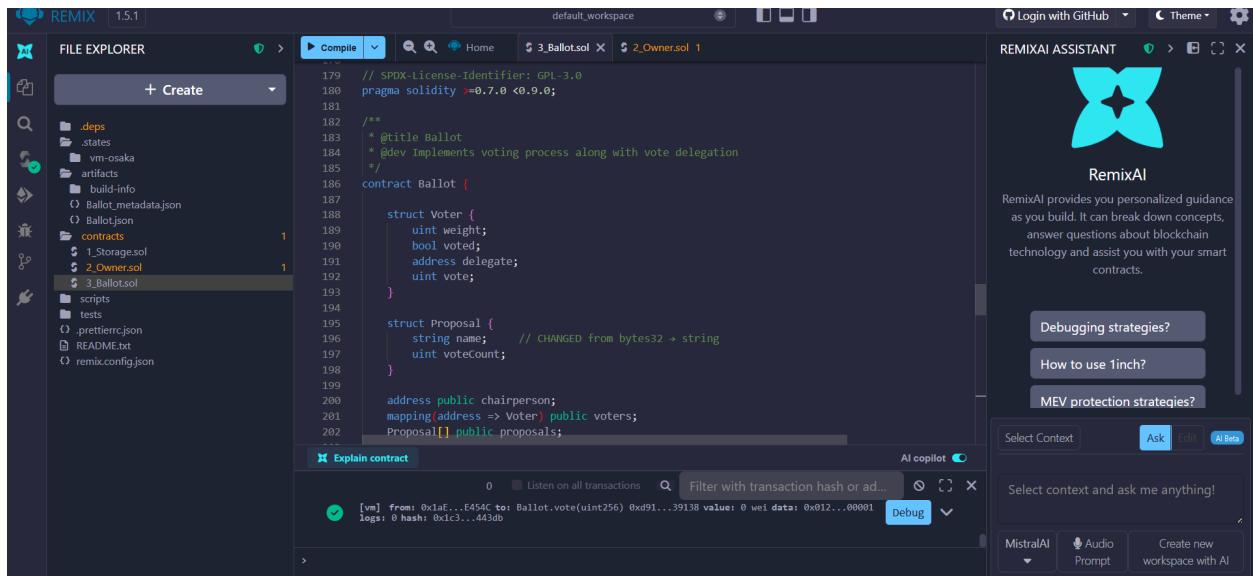
Go to  <https://remix.ethereum.org>

No install needed.

Step 2: Create the file

- In File Explorer

Ballot.sol



Step 3: Compile the contract

1. Open Solidity Compiler (left sidebar)
2. Click Compile Ballot.sol
3. Make sure green tick appears (no errors)

Step 4: Deploy the contract

1. Open Deploy & Run Transactions

2. Set:

- **Environment → Remix VM**
- **Account → keep default**
- **Contract → Ballot**

The screenshot shows the Remix IDE version 1.5.1. On the left, the Solidity Compiler interface is visible with the contract name 'Ballot' selected. In the center, the contract code for '3_Ballot.sol' is displayed, defining a 'Ballot' contract with a 'Voter' struct and various functions. On the right, the 'REMIXAI ASSISTANT' panel is open, featuring a 'RemixAI' logo and a brief description of its features: providing personalized guidance, breaking down concepts, answering questions about blockchain technology, and assisting with smart contracts. It also includes sections for 'Debugging strategies?', 'How to use 1inch?', 'MEV protection strategies?', and an 'Ask' button.

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

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
    }
    mapping(address => Voter) voters;
    Proposal[] public proposals;
}

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
    }
    mapping(address => Voter) voters;
    Proposal[] public proposals;
}
```

Step 5: Enter constructor input (IMPORTANT)

Your constructor is:

`constructor(string[] memory proposalNames)`

So input must be a string array ↗

Example:

`["PranavP", "Bob", "Charlie"]`

```

179 // SPDX-License-Identifier: GPL-3.0
180 pragma solidity >=0.7.0 <0.9.0;
181
182 /**
183 * @title Ballot
184 * @dev Implements voting process along with vote delegation
185 */
186 contract Ballot {
187
188     struct Voter {
189         uint weight;
190         bool voted;
191         address delegate;
192         uint vote;
193     }
194
195     struct Proposal {
196         string name; // CHANGED from bytes32 + string
197         uint voteCount;
198     }
199
200     address public chairperson;
201     mapping(address => Voter) public voters;
202     Proposal[] public proposals;
203 }

```

Step 6: Click Deploy

- **Click Deploy**
- **Contract appears under Deployed Contracts** 🎉
- **The deployer is automatically the chairperson**

giveRightToVote (address voter)

What to add:

Paste a Remix account address (NOT the chairperson).

Important:

- You must be using Account 1 (chairperson) when clicking this
- Switch account at the top if needed

Click Transact

The screenshot shows the Remix IDE version 1.5.1. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, showing a selected account 'Remix VM (Osaka)' with address 0xAb8...35cb2 and 100.0 ETH balance. Below it, there's a 'GAS LIMIT' section with 'Estimated Gas' selected and a 'Custom' value of 3000000 Wei. The 'CONTRACT' section shows the deployed contract 'Ballot - contracts/3_Ballot.sol'. At the bottom of the sidebar, there are 'Deploy' and 'At Address' buttons, and a dropdown menu for 'Deploy to' with values 'PranavP', 'Bob', and 'Charlie'. The main area displays the Solidity code for the Ballot contract:

```

179 // SPDX-License-Identifier: GPL-3.0
180 pragma solidity >0.7.0 <0.9.0;
181 /**
182 * @title Ballot
183 * @dev Implements voting process along with vote delegation
184 */
185 contract Ballot {
186     struct Voter {
187         uint weight;
188         bool voted;
189         address delegate;
190         uint256
191     }
192     struct bytes32 + string
193     str
194     uint voteCount;
195     mapping(address => Voter) public voters;
196     Proposal[] public proposals;
197 }
198
199
200
201
202

```

Below the code, there's an 'Explain contract' button and a transaction log entry: '[vm] from: 0x583...edd4 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0 hash: 0xf85...03a21'. The right side of the interface includes tabs for 'Compile', 'Run', 'Home', and 'Storage', along with a search bar and other navigation elements.

Then vote

2 vote (uint256 proposal)

What to add:

A number, based on proposal index:

0 → PranavP

1 → Bob

2 → Charlie

Example:

0

Important:

- **Switch to the voter account (the one you gave rights to)**
- **Then click Transact**

The screenshot shows the REMIX IDE interface with the Ballot contract deployed. The sidebar displays the deployed contract details, including the address (BALLOT AT 0xD91...39138) and various interaction functions like delegate, giveRightToVote, vote, chairperson, proposals, voters, winnerName, and winningProposal. The main panel shows the Solidity code for the Ballot contract, which includes Voter and Proposal structs, and implements the voting process with delegation. A transaction history section at the bottom indicates a recent transaction from 0xAb8...35cb2 to the contract's address.

```
// 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;
}
```

This screenshot is nearly identical to the one above, showing the REMIX IDE with the Ballot contract deployed. The difference is in the transaction history, where a new transaction is shown: [vm] from: 0xAb8...35cb2 to: Ballot.vote(uint256) 0xd91...39138 value: 0 wei data: 0x012...00000 logs: 0 hash: 0xc66...d2510. This indicates that a vote has been cast for a proposal.

```
// 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;
}
```

```

// 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;
}

```

proposals (uint256)

What to add:

Proposal index:

0

Returns proposal details.

```

// 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;
}

0x [call] from: 0x48209930c481177ec7e8f571cecaE8A9e22C02db to: Ballot.winnerName() data: 0xe2b...a53f0

```

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.