

**Aim:** Deploying a Voting/Ballot Smart Contract

**Lab Objectives:** To explore Blockchain concepts.

**Lab Outcomes (LO):** Design Smart Contract using Solidity (LO2)

**Task to be performed :**

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:**

**What is Remix IDE ?**

Remix IDE is a no-setup tool with a GUI for developing smart contracts. Used by experts and beginners alike, Remix will get you going in double time. Remix plays well with other tools, and allows for a simple deployment process to the chain of your choice. Remix is famous for its visual debugger. Remix is the place everyone comes to learn Ethereum.

**Solidity:**

Solidity is a brand-new programming language created by Ethereum which is the second-largest market of cryptocurrency by capitalization, released in the year 2015 and led by Christian Reitwiessner. Some key features of solidity are listed below:

- Solidity is a high-level programming language designed for implementing smart contracts.
- It is a statically typed object-oriented(contract-oriented) language.
- Solidity is highly influenced by Python, c++, and JavaScript which run on the Ethereum Virtual Machine(EVM).
- Solidity supports complex user-defined programming, libraries, and inheritance.
- Solidity is the primary language for blockchains running platforms.
- Solidity can be used to create contracts like voting, blind auctions, crowdfunding, multi-signature wallets, etc.

- **Version Declaration:** In every Solidity contract, you should specify the compiler version you intend to use. This is done using the ``pragma`` statement. For example:

```
``solidity
    pragma solidity ^0.8.0;
``
```

This statement indicates that the contract should be compiled using a Solidity compiler version greater than or equal to 0.8.0 but less than 0.9.0.

- **Contract Declaration:** A Solidity contract is similar to a class in other programming languages. It encapsulates the code and data for a smart contract. Here's a basic contract declaration:

```
``solidity
contract MyContract {
    // State variables and functions go here
}
``
```

- **State Variables:** State variables represent the data stored in a smart contract. They persist across function calls and transactions.  
For example:

```
``solidity
uint256 public myValue;
``
```

In this example, ``myValue`` is a state variable of type ``uint256``, and it's marked as ``public``, which means it can be accessed from outside the contract.

- **Functions:** Functions in Solidity define the behavior and logic of your smart contract.

```
``solidity
function setValue(uint256 newValue) public {
    myValue = newValue;
}
``
```

This function takes an `uint256` parameter `newValue` and sets the `myValue` state variable to the new value.

- **Visibility Modifiers:** Functions and state variables can have visibility modifiers that control who can access them. The common modifiers are:

`public`: Anyone can access.

`private`: Only the contract itself can access.

`internal`: Accessible from within the contract and derived contracts.

`external`: Only accessible externally, i.e., from outside the contract.

- **Constructor:** The constructor is a special function that gets executed once when the contract is deployed. It is used to initialize state variables and perform any setup tasks.

```
```solidity
constructor() {
    myValue = 0;
}
```
```

- **Events:** Events are used to log and store information about specific occurrences in your contract. They are often used for debugging and to provide information to external systems.

```
```solidity
event ValueSet(uint256 indexed newValue);

function setValue(uint256 newValue) public {
    myValue = newValue;
    emit ValueSet(newValue);
}
```
```

- **Compile and Deploy:** After writing your contract in Remix or any other Solidity development environment, you can compile it to generate bytecode. Then, you can deploy it to the Ethereum blockchain. In Remix, you can deploy and interact with your contract using the built-in Remix VM or by connecting to a real Ethereum network.

These are the fundamental concepts of Solidity. As you become more familiar with the language, you can explore more advanced topics like inheritance, modifiers, libraries, and more to build complex and secure smart contracts. Solidity has a rich ecosystem and documentation to help you learn and develop blockchain applications

## **What is the relevance of require statements in the functions of Solidity Programs?**

In Solidity, require statements are used within functions to enforce conditions or constraints that must be satisfied for the function to proceed. They play a crucial role in ensuring the correctness and security of smart contracts. Here's the relevance and purpose of require statements in Solidity programs:

**1. Input Validation:** Require statements are often used to validate input parameters to functions. They ensure that the inputs meet specific criteria before allowing the function to continue execution. For example, you might require that an address parameter is not null or that an integer parameter is greater than zero.

**2. Preconditions:** Require statements establish preconditions that must be met for the function to execute. If these conditions are not satisfied, the function will revert, and any changes made to the contract's state during the function's execution will be rolled back. This helps prevent invalid state changes and protects against malicious or unintended actions.

**3. Security:** Require statements are an essential part of securing smart contracts. They help prevent common vulnerabilities like integer overflow, division by zero, and unauthorized access. By checking conditions explicitly, you reduce the attack surface of your contract and make it less susceptible to exploitation.

**4. Gas Consumption:** When a require statement fails (i.e., the condition is not met), it causes the transaction to revert, and any gas used up to that point is refunded to the sender. This prevents unnecessary gas expenditure in the case of invalid operations, which is cost-effective for users.

## **Why bytes32 instead of string?**

In Solidity, you often see `bytes32` used instead of `string` for several reasons, primarily related to efficiency, gas costs, and data storage on the Ethereum blockchain. Here are the main considerations:

**Gas Costs:** Storing and manipulating `string` data in Ethereum smart contracts is more expensive in terms of gas costs compared to using `bytes32`. Gas is the unit of computation on the Ethereum network, and each operation in a smart contract consumes gas. Storing and processing variable-length data like strings requires more gas because the size of the data can vary significantly.

**Fixed Size:** `bytes32` is a fixed-size data type, which means it always occupies 32 bytes of storage, regardless of the content's length. In contrast, the size of a `string` depends on the length

of the string, making it more challenging to predict storage requirements and associated gas costs.

**EVM Limitations:** The Ethereum Virtual Machine (EVM) has limitations on how much gas can be consumed in a single transaction. Storing or processing very long strings in a single transaction can exceed these gas limits, causing the transaction to fail.

**Data Alignment:** `bytes32` is aligned to 32-byte boundaries in memory, making it more efficient for the EVM to access and manipulate. On the other hand, strings can have variable lengths, which may not align well with EVM memory management, potentially leading to higher gas costs.

**Compatibility:** Some Ethereum tools and libraries work more smoothly with `bytes32` than with `string`. For example, when interacting with other smart contracts or using off-chain tools to analyze on-chain data, `bytes32` can be more compatible and straightforward to work with.

However, it's important to note that using `bytes32` has limitations. It's not suitable for storing long or human-readable text, as it is limited to 32 bytes and is not designed for natural language or user interface purposes. For most use cases involving user input or text that needs to be human-readable, `string` or `bytes` (for variable-length binary data) is the appropriate choice.

### **Code:**

```
// SPDX-License-Identifier: GPL-3.0
```

```
pragma solidity >=0.7.0 <0.9.0;
```

```
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 {
        // 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;

mapping(address => Voter) public voters;

Proposal[] public proposals;

constructor(bytes32[] 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) public {
    require(
        msg.sender == chairperson,
        "Only chairperson can give right to vote."
    );
    require(
        !voters[voter].voted,
        "The voter already voted."
    );
    require(voters[voter].weight == 0);
    voters[voter].weight = 1;
}

function delegate(address to) public {
    Voter storage sender = voters[msg.sender];
    require(!sender.voted, "You already voted.");
    require(to != msg.sender, "Self-delegation is disallowed.");
}

```

```

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.");
}
sender.voted = true;
sender.delegate = to;
Voter storage delegate_ = voters[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;
}
}

```

```

function vote(uint proposal) public {
    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;
}

```

```

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;

```

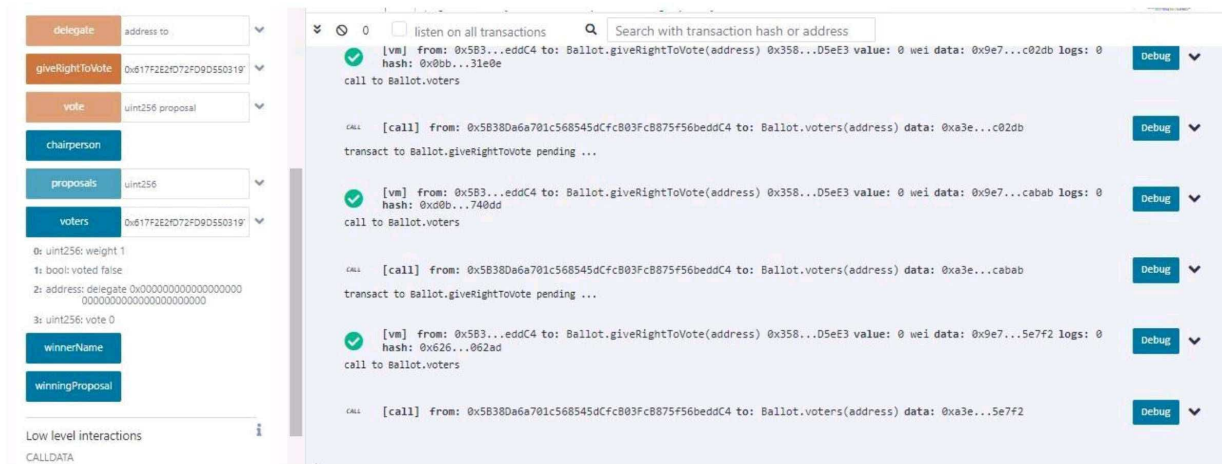
```
        winningProposal_ = p;
    }
}

function winnerName() public view
    returns (bytes32 winnerName_)
{
    winnerName_ = proposals[winningProposal()].name;
}
}
```



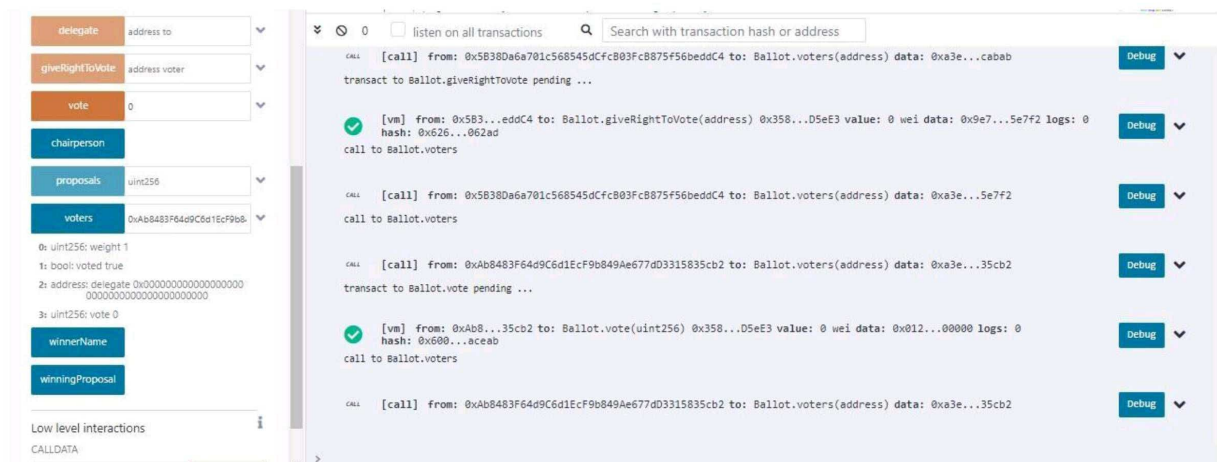
## Output:

### Giving right to voters



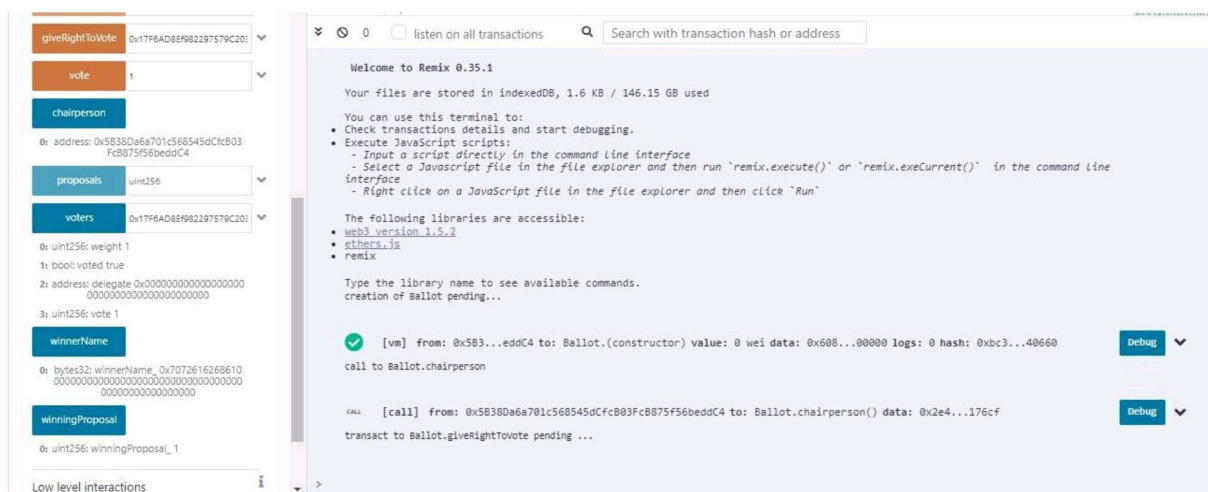
The screenshot shows the Remix IDE interface. On the left, the 'Ballot' contract is loaded, and the 'giveRightToVote' function is selected. The main area displays the transaction details for the 'giveRightToVote' function call, showing the transaction hash and the data being passed to the function.

### After voting



The screenshot shows the Remix IDE interface. On the left, the 'Ballot' contract is loaded, and the 'vote' function is selected. The main area displays the transaction details for the 'vote' function call, showing the transaction hash and the data being passed to the function.

### Winner name






The screenshot shows the Remix IDE interface. On the left, the 'Ballot' contract is loaded, and the 'winnerName' function is selected. The main area displays the transaction details for the 'winnerName' function call, showing the transaction hash and the data being passed to the function.



## Code: (string)

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  pragma solidity >=0.7.0 <0.9.0;
4
5  /**
6   * @title Ballot
7   * @dev Implements voting process along with vote delegation
8   */
9  contract Ballot {
10
11     struct Voter {
12         uint weight; // weight is accumulated by delegation
13         bool voted; // if true, that person already voted
14         address delegate; // person delegated to
15         uint vote; // index of the voted proposal
16     }
17
18     struct Proposal {
19         // If you can limit the length to a certain number of bytes,
20         // always use one of bytes1 to bytes32 because they are much cheaper
21         string name; // short name (up to 32 bytes)
22         uint voteCount; // number of accumulated votes
23     }
24
25     address public chairperson;
26
27     mapping(address => Voter) public voters;
28
29     Proposal[] public proposals;
30
31     /**
32     * @dev Create a new ballot to choose one of 'proposalNames'.
33     * @param proposalNames names of proposals
34     */
35     constructor(string[] memory proposalNames) {
36         chairperson = msg.sender;
37         voters[chairperson].weight = 1;
38
39         for (uint i = 0; i < proposalNames.length; i++) {
40             // 'Proposal({...})' creates a temporary
41             // Proposal object and 'proposals.push(...)'
42             // appends it to the end of 'proposals'.
43             proposals.push(Proposal({
44                 name: proposalNames[i],
45                 voteCount: 0
46             }));
47         }
48     }
49
50     /**
51     * @dev Give 'voter' the right to vote on this ballot. May only be called by 'chairperson'.
52     * @param voter address of voter
53     */
54     function giveRightToVote(address voter) public {
55         require(
56             msg.sender == chairperson,
57             "Only chairperson can give right to vote."
58         );
59         require(
60             !voters[voter].voted,
```

```

61         "The voter already voted."
62     );
63     require(voters[voter].weight == 0);
64     voters[voter].weight = 1;
65 }
66
67 /**
68  * @dev Delegate your vote to the voter 'to'.
69  * @param to address to which vote is delegated
70  */
71 function delegate(address to) public {  infinite gas
72     Voter storage sender = voters[msg.sender];
73     require(!sender.voted, "You already voted.");
74     require(to != msg.sender, "Self-delegation is disallowed.");
75
76     while (voters[to].delegate != address(0)) {
77         to = voters[to].delegate;
78
79         // We found a loop in the delegation, not allowed.
80         require(to != msg.sender, "Found loop in delegation.");
81     }
82     sender.voted = true;
83     sender.delegate = to;
84     Voter storage delegate_ = voters[to];
85     if (delegate_.voted) {
86         // If the delegate already voted,
87         // directly add to the number of votes
88         proposals[delegate_.vote].voteCount += sender.weight;
89     } else {
90         delegate_.weight += sender.weight;
91     }
92 }
93
94 }
95
96 /**
97  * @dev Give your vote (including votes delegated to you) to proposal 'proposals[proposal].name'.
98  * @param proposal index of proposal in the proposals array
99  */
100 function vote(uint proposal) public {  infinite gas
101     Voter storage sender = voters[msg.sender];
102     require(sender.weight != 0, "Has no right to vote");
103     require(!sender.voted, "Already voted.");
104     sender.voted = true;
105     sender.vote = proposal;
106
107     // If 'proposal' is out of the range of the array,
108     // this will throw automatically and revert all
109     // changes.
110     proposals[proposal].voteCount += sender.weight;
111 }
112
113 /**
114  * @dev Computes the winning proposal taking all previous votes into account.
115  * @return winningProposal_ index of winning proposal in the proposals array
116  */
117 function winningProposal() public view  infinite gas
118     returns (uint winningProposal_)
119 {
120     uint winningVoteCount = 0;
121     for (uint p = 0; p < proposals.length; p++) {

```

```

122         if (proposals[p].voteCount > winningVoteCount) {
123             winningVoteCount = proposals[p].voteCount;
124             winningProposal_ = p;
125         }
126     }
127 }
128
129 /**
130  * @dev Calls winningProposal() function to get the index of the winner contained in the proposals array and then
131  * @return winnerName_ the name of the winner
132  */
133 function winnerName() public view {
134     returns (string memory winnerName_)
135     {
136         winnerName_ = proposals[winningProposal()].name;
137     }
138 }

```

## Output: Winner name

The screenshot displays a web application interface for a voting system and its corresponding transaction log. The interface on the left includes a 'vote' button, a 'chairperson' dropdown, a 'proposals' dropdown, and a 'winners' dropdown. Below these are input fields for 'weight', 'address', 'vote', 'winnerName', and 'winningProposal', each with a corresponding button. The transaction log on the right shows a sequence of events: a successful 'vote' transaction, a successful 'winnerName' call, and a successful 'winningProposal' call. The log also shows a 'revert' message with the reason 'Has no right to vote'.

**Conclusion:** In this experiment, we understood the logic of the Ballot contract in Solidity. We successfully performed the deployment of the contract. Also implemented the deployment of the contract by converting bytes32 to string.