PRISHA

1: VOTING

First of all this is a smart contract that implements voting , the name of the contract is ballot.

The contract begins with SPDX License identifier and pragma statement specifying the licence and version it is compatible with.

```
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 struct defines the structure of the voter. It has four properties :
WEIGHT : represents voting power
VOTED: to keep check if the person has already voted (to prevent multiple votes)
DELEGATE: Address of person the voter has delegated
VOTE: represents the index of voted proposal to ensure uniqueness

```
// This is a type for a single proposal.
struct Proposal {
    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;
```

This part of code define the structure of proposal it has two properties
Name: short name of proposal
Votecount: represents the number of votes accumulated for that proposal
After this a public state variable chair person having type address is declared(

ig its the one who deployed the contract)

So there is then this mapping named vectors that maps address to voter

structs

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

/// Create a new ballot to choose one of `proposalNames`
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
        }));
    }
}
```

Initially a dynamically sized array is created with the name proposals to store the list of proposals then there is this constructor function which gets executed when the contract is deployed . It iterates over all the proposal names which are passed in argument and creates a struct for each of them assigning vote count to 0.

All the structs are being stored in a proposals array.

```
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, bu
    // 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);
    voters[voter].weight = 1;
}
```

It allows the chairperson to give a voter the right to vote. It checks first via require if the caller is the chairperson then checks if their voter hasn't voted and has no weight , then it is assigned weight of 1. So in short after passing, require checks the voter if not voted is assigned weight.

```
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;
    }
}
```

So this delegate function as far i understood allows a voter to delegate vote to another voter who is given by to address. First check if it has the right to vote and has not already voted and also they are not delegating to people who are not even standing lol . After the checks pass the senders voted status is updated to voted and his weight is added to the chosen proposals vote count yay vote done

```
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
```

This is the function that allows voters to cast their vote after checking that they haven't already voted and also they have the right to vote as given by the chairman in the function already discussed . After this the vote count of the voted proposal is increased. By now yes the voters have voted time for results

```
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;
        }
    }
}

// Calls winningProposal() function to get the index
// of the winner contained in the proposals array and ther
// returns the name of the winner
function winnerName() external view
        returns (bytes32 winnerName_)
{
    winnerName_ = proposals[winningProposal()].name;
}
```

So it's time for the result , it is done by iterating through the array storing the index of the proposal which gate the highest vote updating it as soon as a value greater than max is there .

After this the winner 's name is declared using the index we stored above .

So that was all my understanding of the proposal!!