Blockchain Study Notes Day 16:

Module 3 - Solidity Advanced
Chapter 2 - Structs in Solidity

Introduction to Structs

Structs in Solidity are used to define custom data types that group multiple related variables under a single type. They enable developers to model more complex data structures within smart contracts.

---

1. What Are Structs?

- **Definition**:
  Structs allow the creation of custom data types that group multiple variables, each potentially of a different type.
- **Purpose**:
  - Improve code organization.
  - Enable complex data modeling.

---

2. Syntax for Structs

Defining a Struct:

```
struct StructName {
    uint id;
    string name;
    bool isActive;
}
```

Declaring a Struct Variable:

```
StructName public myStruct;
```

Initializing a Struct:

```
myStruct = StructName(1, "Munawar", true);
```

---

3. Example Program Demonstrating Structs (Using Munawar)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MunawarStructs {
    // Define a struct to represent a user
    struct User {
        uint id;
        string name;
        bool isActive;
    }

    // Mapping to store user information by ID
    mapping(uint => User) public users;

    // Function to create a new user
    function createUser(uint _id, string memory _name, bool _isActive) public
{
        users[_id] = User(_id, _name, _isActive);
    }

    // Function to get user information by ID
    function getUser(uint _id) public view returns (User memory) {
        return users[_id];
    }

    // Function to update a user's active status
    function updateUserStatus(uint _id, bool _isActive) public {
        users[_id].isActive = _isActive;
    }
}
```

---

4. Operations on Structs

4.1. Initializing Structs in Different Ways

- **Using Constructor Style**:

  ```
  User memory newUser = User(1, "Munawar", true);
  ```

- **Key-Value Initialization**:

  ```
  User memory newUser = User({ id: 1, name: "Munawar", isActive: true });
  ```

4.2. Updating Struct Fields

- Update specific fields directly:

  ```
  users[_id].name = "Updated Name";
  ```

4.3. Deleting Struct Data

- Delete a struct entry:

```
delete users[_id];
```

---

5. Advanced Struct Usage

5.1. Arrays of Structs

- Useful for maintaining a list of struct instances.
- **Example**:

```
User[] public userList;

function addUserToList(uint _id, string memory _name, bool _isActive)
public {
    userList.push(User(_id, _name, _isActive));
}
```

5.2. Nested Structs

- Structs can contain other structs as fields.
- **Example**:

```
struct Profile {
    uint age;
    string bio;
}

struct User {
    uint id;
    string name;
    Profile profile;
}
```

---

6. Best Practices for Structs

- **Efficient Data Storage**:
  - Avoid storing unnecessary data in structs to minimize gas costs.
- **Use Memory for Temporary Structs**:
  - Use `memory` keyword for temporary structs in functions to save gas.
- **Avoid Deep Nesting**:
  - Limit nested structs to maintain code readability and reduce complexity.

Home Task

1. **Extend the Example Program**:
    - Add a function to deactivate all users in the `userList`.
2. **Create a New Contract**:
    - Implement a contract that models a product catalog using structs, with fields for `productId`, `productName`, and `price`.
3. **Research**:
    - Explore real-world applications where structs are used for complex data modeling in Solidity.

Conclusion

Structs in Solidity are a powerful tool for defining custom data types and managing complex data structures. By effectively using structs, developers can improve the organization, readability, and functionality of their smart contracts.

.

Day 16 Notes

Prepared by Munawar Johar