1. **Data Structure Selection:**

Decision: Utilize a mapping data structure to store contract addresses and their corresponding descriptions. The mapping provides efficient key-value storage and retrieval operations.

Justification: Mapping allows for constant-time access to contract addresses and descriptions, ensuring efficient management of storage and updates.

2. **Efficient Removal Strategy:**

Decision: Implement a logical deletion approach for contract removal rather than physically deleting data. Use a flag or status variable to mark contracts as inactive.

Justification: Logical deletion prevents gaps or unused data in storage, maintaining data integrity and optimizing storage usage. It also simplifies the removal process by avoiding costly data relocation operations.

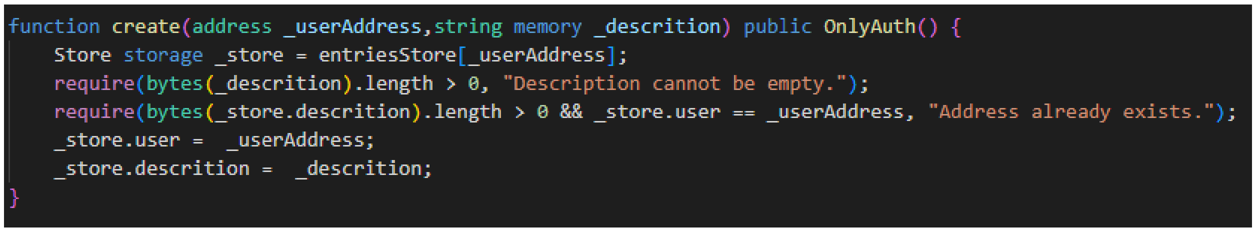
3. **Access Control Mechanism:**

Decision: Implement role-based access control using modifiers to restrict access to specific functions. Define roles such as owner/administrator and regular user. The contract inherits from an Access contract, indicating that access control mechanisms are implemented. The assumption is that the **OnlyAuth** modifier in the functions is defined in the Access contract and ensures that only authorized users can call these functions. However, the details of this access control mechanism are not provided here, so you'd need to define it in the **Access.sol** file.

Justification: Role-based access control ensures that only authorized users can perform critical actions such as adding, updating, or removing contract addresses. Modifiers simplify access control enforcement and enhance code readability.

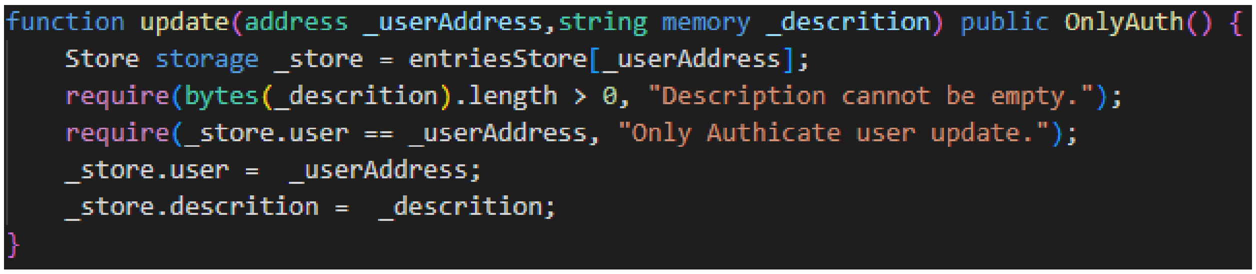
4. **Functionality:**

Create: Allows adding a new entry to the store for a given user address. It requires a non-empty description and ensures that the address doesn't already exist in the store.



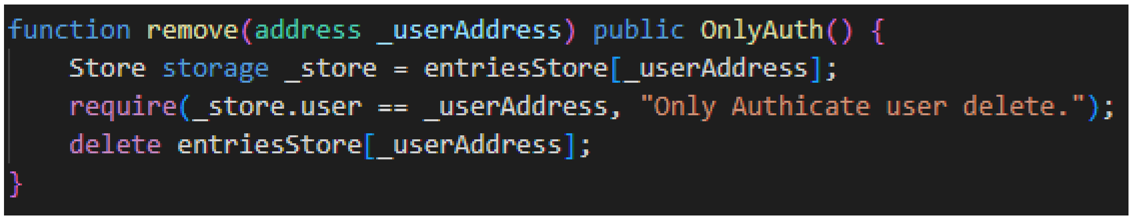
```
function create(address _userAddress,string memory _descrition) public OnlyAuth() {
    Store storage _store = entriesStore[_userAddress];
    require(bytes(_descrition).length > 0, "Description cannot be empty.");
    require(bytes(_store.descrition).length > 0 && _store.user == _userAddress, "Address already exists.");
    _store.user =  _userAddress;
    _store.descrition =  _descrition;
}
```

Update: Allows updating the description for an existing user entry. It requires a non-empty description and ensures that the caller is authorized and the user address exists in the store.



```
function update(address _userAddress,string memory _descrition) public OnlyAuth() {
    Store storage _store = entriesStore[_userAddress];
    require(bytes(_descrition).length > 0, "Description cannot be empty.");
    require(_store.user == _userAddress, "Only Authicate user update.");
    _store.user =  _userAddress;
    _store.descrition =  _descrition;
}
```

Remove: Allows removing an entry from the store for a given user address. It ensures that the caller is authorized and the user address exists in the store.



```
function remove(address _userAddress) public OnlyAuth() {
    Store storage _store = entriesStore[_userAddress];
    require(_store.user == _userAddress, "Only Authicate user delete.");
    delete entriesStore[_userAddress];
}
```

5. **Assumptions:**

It's assumed that the OnlyAuth modifier is correctly implemented in the Access contract to enforce access control.

It's assumed that the Access contract defines who has authorization to call the functions of this contract.

It's assumed that only the owner of a user entry can update or remove it.

It's assumed that the descrition (typo in the variable name, should be description) is sufficient to describe the user's entry, but depending on the application, additional fields may be required.

It's assumed that there is no need for events to log the actions performed on the contract data. If logging is necessary for audit purposes, events should be added accordingly.