Solidity Storage vs Memory Quiz

Quiz 1: Understanding Storage vs Memory Basics

Instructions: Neri is optimizing her smart contract to counter Hackana's data attacks. Which statement correctly describes the difference between storage and memory in Solidity?

```
pragma solidity ^0.8.0;

contract DataLocations {
    // State variable stored in storage
    string public storedData = "Persistent data";

    function compareLocations() public view returns (bool) {
        // Local variable in memory
        string memory tempData = "Temporary data";

        // Various statements about data locations
        // A: Storage is temporary, memory is permanent
        // B: Both storage and memory persist after function execution
        // C: Storage is permanent on the blockchain, memory exists only during
function execution
        // D: Memory is cheaper to use but can't be modified

        return true; // Just a placeholder
    }
}
```

Which statement correctly describes storage and memory in Solidity?

- A) Storage is temporary, memory is permanent
- B) Both storage and memory persist after function execution
- C) Storage is permanent on the blockchain, memory exists only during function execution
- D) Memory is cheaper to use but can't be modified

Answer: C) Storage is permanent on the blockchain, memory exists only during function execution

Explanation: Option C correctly describes the fundamental difference between storage and memory in Solidity:

- **Storage**: Data stored in storage (like the `storedData` variable in the example) is permanently saved on the blockchain. It persists between function calls and transactions. However, writing to storage is expensive in terms of gas costs.

- **Memory**: Data in memory (like the `tempData` variable) only exists during the execution of a function. Once the function completes, memory variables are discarded. Memory is cheaper to use but temporary.

Understanding this distinction is crucial for optimizing gas costs and ensuring data persistence when needed.

Quiz 2: Gas Costs of Storage vs Memory

Instructions: Neri needs to optimize her smart contract for gas efficiency. Which function uses data locations most efficiently for gas costs?

```
pragma solidity ^0.8.0;

contract GasOptimization {
    struct Person {
        string name;
        uint256 age;
    }

    // Array of Person structs stored in storage
    Person[] public people;

    // Option A
    function addPerson1(string memory _name, uint256 _age) public {
        Person storage newPerson = people.push();
        newPerson.name = _name;
        newPerson.age = _age;
    }

    // Option B
    function addPerson2(string memory _name, uint256 _age) public {
        Person memory newPerson = Person(_name, _age);
        people.push(newPerson);
    }

    // Option C
    function updateAge1(uint256 index, uint256 newAge) public {
        Person storage personToUpdate = people[index];
        personToUpdate.age = newAge;
    }

    // Option D
    function updateAge2(uint256 index, uint256 newAge) public {
        Person memory personToUpdate = people[index];
        personToUpdate.age = newAge;
    }
}
```

Which function uses data locations most efficiently for gas costs?

- A) addPerson1()
- B) addPerson2()
- C) updateAge1()
- D) updateAge2()

Answer: C) updateAge1()

Explanation: Function `updateAge1()` is the most gas-efficient because:

- It uses `storage` to directly reference the person in the storage array without creating a separate copy
- It only modifies the specific field that needs changing
- The change is directly made to storage without unnecessary copies

The other functions have inefficiencies:

- `addPerson1()` uses storage properly but has an older approach to array pushing
- `addPerson2()` creates an unnecessary memory copy before pushing to storage
- `updateAge2()` creates a memory copy of the Person but doesn't actually update the storage version - the change is lost when the function exits

When updating existing storage data, using a storage reference (like in `updateAge1()`) is the most gas-efficient approach.

Quiz 3: Storage vs Memory with Arrays

Instructions: Neri is creating a function to process barangay supply data. Which function correctly handles arrays based on their data location?

```
pragma solidity ^0.8.0;

contract SupplyTracker {
    uint256[] public supplies = [100, 200, 300];

    // Option A
    function processSupplies1() public view returns (uint256) {
        uint256[] storage supplyData = supplies;
        supplyData[0] = 150; // Try to modify the first element
        return supplies[0];
    }

    // Option B
    function processSupplies2() public view returns (uint256) {
        uint256[] memory supplyData = supplies;
        supplyData[0] = 150; // Try to modify the first element
        return supplies[0];
    }

    // Option C
    function processSupplies3() public returns (uint256) {
        uint256[] storage supplyData = supplies;
        supplyData[0] = 150; // Try to modify the first element
        return supplies[0];
    }

    // Option D
    function processSupplies4() public returns (uint256) {
        uint256[] memory supplyData = supplies;
        supplyData[0] = 150; // Try to modify the first element
        return supplies[0];
    }
}
```

Which function will return 150 after execution?

- A) processSupplies1()
- B) processSupplies2()
- C) processSupplies3()
- D) processSupplies4()

Answer: C) processSupplies3()

Explanation: Function `processSupplies3()` will return 150 because:

1. It creates a storage reference `supplyData` that points to the `supplies` storage array
2. Modifying `supplyData[0]` actually modifies `supplies[0]` since they reference the same storage location
3. The function is not marked as `view` or `pure`, allowing state modifications

The other functions won't work as expected because:

- `processSupplies1()` is marked as `view` which doesn't allow state modifications, so it will fail
- `processSupplies2()` creates a memory copy of the supplies array, so modifying it doesn't affect the original array
- `processSupplies4()` also creates a memory copy which doesn't affect the original storage array

This demonstrates how storage references allow direct modification of state variables while memory creates a separate copy.

Quiz 4: Memory vs Storage for Structs

Instructions: Neri is developing a citizen registry system. Identify which function correctly handles struct data based on their memory requirements.

```
pragma solidity ^0.8.0;

contract CitizenRegistry {
    struct Citizen {
        string name;
        uint256 id;
        bool isActive;
    }

    mapping(uint256 => Citizen) public citizens;

    function registerCitizen(uint256 _id, string memory _name) public {
        citizens[_id] = Citizen(_name, _id, true);
    }
    // Option A
    function updateCitizenStatus1(uint256 _id, bool _status) public {
        Citizen storage citizen = citizens[_id];
        citizen.isActive = _status;
    }

    // Option B
    function updateCitizenStatus2(uint256 _id, bool _status) public {
        Citizen memory citizen = citizens[_id];
        citizen.isActive = _status;
    }

    // Option C
    function getCitizenName1(uint256 _id) public view returns (string memory) {
        Citizen storage citizen = citizens[_id];
        return citizen.name;
    }

    // Option D
    function getCitizenName2(uint256 _id) public view returns (string memory) {
        Citizen memory citizen = citizens[_id];
        return citizen.name;
    }
}
```

Which pair of functions correctly uses data locations for their intended purposes?

- A) updateCitizenStatus1() and getCitizenName1()
- B) updateCitizenStatus1() and getCitizenName2()
- C) updateCitizenStatus2() and getCitizenName1()
- D) updateCitizenStatus2() and getCitizenName2()

Answer: B) updateCitizenStatus1() and getCitizenName2()

Explanation: Option B represents the optimal use of storage and memory:

- `updateCitizenStatus1()` correctly uses `storage` to update a citizen's status in the mapping. Using storage is necessary when modifying state.

- `getCitizenName2()` correctly uses `memory` for a read-only operation that doesn't modify state. Since the function only reads data, using memory is more gas-efficient.

The other combinations are suboptimal:

- Option A uses storage in a read-only function, which works but is less gas-efficient
- Option C uses memory in `updateCitizenStatus2()` which means the status update doesn't actually get stored
- Option D uses memory in both functions, which means the status update in `updateCitizenStatus2()` doesn't persist

When modifying state, use storage references. For read-only operations, prefer memory for better gas efficiency.