Solidity Structs Quiz

Quiz 1: Understanding Structs

Instructions: Neri is creating a system to track user information to fight against Hackana. Which of these correctly defines a `User` struct in Solidity?

```
pragma solidity ^0.8.0;

contract UserRegistry {
    // Option A
    struct User {
        uint256 id;
        string name;
        address walletAddress;
        bool isActive;
    }

    // Option B
    User struct {
        uint256 id;
        string name;
        address walletAddress;
        bool isActive;
    }

    // Option C
    class User {
        uint256 id;
        string name;
        address walletAddress;
        bool isActive;
    }

    // Option D
    type User = {
        uint256 id;
        string name;
        address walletAddress;
        bool isActive;
    }
}
```

Which option correctly defines a struct in Solidity?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: A) Option A

Explanation: Option A correctly defines a struct in Solidity. The proper syntax is:

```
struct StructName {
    Type1 fieldName1;
    Type2 fieldName2;
    // etc.
}
```

This struct creates a custom data type called `User` that bundles together related pieces of information: an ID, a name, a wallet address, and an active status. It's like creating a form with different fields to store information about a single user in one organized package. Options B, C, and D use incorrect syntax that doesn't exist in Solidity.

Quiz 2: Creating and Using Structs

Instructions: Neri wants to create a new product record in her inventory system. Which option correctly creates a new `Product` struct and assigns values to it?

```
pragma solidity ^0.8.0;

contract InventorySystem {
    struct Product {
        uint256 id;
        string name;
        uint256 price;
        uint256 quantity;
    }

    mapping(uint256 => Product) public inventory;

    // Option A
    function addProduct1(uint256 _id, string memory _name, uint256 _price, uint256
_quantity) public {
        Product memory newProduct = Product(_id, _name, _price, _quantity);
        inventory[_id] = newProduct;
    }

    // Option B
    function addProduct2(uint256 _id, string memory _name, uint256 _price, uint256
_quantity) public {
        inventory[_id] = Product{id: _id, name: _name, price: _price, quantity:
_quantity};
    }

    // Option C
    function addProduct3(uint256 _id, string memory _name, uint256 _price, uint256
_quantity) public {
        inventory[_id] = new Product(_id, _name, _price, _quantity);
```

```
    }

    // Option D
    function addProduct4(uint256 _id, string memory _name, uint256 _price, uint256
_quantity) public {
        inventory[_id].id = _id;
        inventory[_id].name = _name;
        inventory[_id].price = _price;
        inventory[_id].quantity = _quantity;
    }
}
```

Which option correctly creates a Product struct and adds it to inventory?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: A) Option A

Explanation: Option A correctly creates a new `Product` struct and adds it to inventory. It follows these steps:

1. Creates a new struct in memory using `Product memory newProduct = Product(_id, _name, _price, _quantity)`
2. Assigns this struct to the mapping using the product ID as the key: `inventory[_id] = newProduct`

Option B uses named parameters which is valid in newer Solidity versions, but the syntax is incorrect (it should be `Product({id: _id, ...})` with parentheses and curly braces). Option C incorrectly uses the `new` keyword which is for contracts, not structs. Option D works but is less efficient than Option A since it writes to storage multiple times instead of creating the struct in memory first.

Quiz 3: Storing Structs in Mappings

Instructions: Neri is building a system to track vehicles. Which code correctly sets up a mapping to store Vehicle structs and adds a function to register new vehicles?

```
pragma solidity ^0.8.0;

contract VehicleRegistry {
    // Vehicle struct definition
    struct Vehicle {
        string make;
        string model;
        uint256 year;
        address owner;
    }

    // Option A
    Vehicle[] public vehicles;
```

```
    function registerVehicle1(string memory _make, string memory _model, uint256
_year) public {
        vehicles.push(Vehicle(_make, _model, _year, msg.sender));
    }

    // Option B
    mapping(address => Vehicle) public ownerToVehicle;

    function registerVehicle2(string memory _make, string memory _model, uint256
_year) public {
        ownerToVehicle[msg.sender] = Vehicle(_make, _model, _year, msg.sender);
    }

    // Option C
    mapping(string => Vehicle) public vehiclesByMake;

    function registerVehicle3(string memory _make, string memory _model, uint256
_year) public {
        vehiclesByMake[_make] = Vehicle(_make, _model, _year, msg.sender);
    }

    // Option D
    mapping(address => Vehicle[]) public ownerVehicles;

    function registerVehicle4(string memory _make, string memory _model, uint256
_year) public {
        Vehicle memory newVehicle = Vehicle(_make, _model, _year, msg.sender);
        ownerVehicles[msg.sender].push(newVehicle);
    }
}
```

Which option is best for allowing owners to register multiple vehicles?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: D) Option D

Explanation: Option D is best for allowing owners to register multiple vehicles because it:

1. Creates a mapping that links each owner address to an array of vehicles
2. Uses the `.push()` method to add a new vehicle to that owner's collection

This allows one owner to have many vehicles, which is realistic. Option A stores vehicles in an array but doesn't organize them by owner. Option B only allows one vehicle per owner (new vehicles would overwrite the old one). Option C uses the make (like "Toyota") as the key, which would mean only one vehicle per make could exist in the system.

Quiz 4: Updating Struct Values

Instructions: Neri needs to update information in her user registry. Which function correctly updates a specific field in a User struct?

```
pragma solidity ^0.8.0;

contract UserUpdates {
    struct User {
        string name;
        uint256 age;
        bool isActive;
    }

    mapping(address => User) public users;

    // Register a new user
    function registerUser(string memory _name, uint256 _age) public {
        users[msg.sender] = User(_name, _age, true);
    }

    // Option A
    function updateAge1(uint256 _newAge) public {
        User memory user = users[msg.sender];
        user.age = _newAge;
    }

    // Option B
    function updateAge2(uint256 _newAge) public {
        users[msg.sender].age = _newAge;
    }

    // Option C
    function updateAge3(uint256 _newAge) public {
        User storage user = users[msg.sender];
        user.age = _newAge;
    }

    // Option D
    function updateAge4(uint256 _newAge) public {
        users[msg.sender] = User(users[msg.sender].name, _newAge,
users[msg.sender].isActive);
    }
}
```

Which function correctly updates a user's age?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: C) Option C

Explanation: Options B and C both work correctly, but Option C is more gas-efficient for updating multiple fields. Here's why:

Option C uses `storage` which creates a reference to the struct in storage. This means:

1. Any changes you make to the `user` variable directly change the stored data
2. It's more efficient when changing multiple fields, as you only reference the mapping lookup once

Option B directly changes the field and also works correctly, but would be less efficient if updating multiple fields.

Option A doesn't work because it creates a copy in `memory`, and changes to that copy don't affect the stored struct.

Option D works but is inefficient - it creates an entirely new struct and overwrites the old one, which uses more gas than just updating a field.