LAB 07 August, 2023

1 Mappings

Mappings in Solidity provide a convenient way to store and retrieve data in the form of key-value pairs.

They act as hash tables or dictionaries, associating a unique key with a corresponding value.

*In Solidity, mappings are commonly used to link Ethereum addresses with specific values.

Mappings do not store any key information or length information directly. Instead, they internally use the **keccak256** hash function to calculate the offset of the value based on the key.

It should also be noted that ‘structs’ and ‘enums’ can be mapped as well. However ‘structs’ cannot be used as key values in mapping.

1.1 Syntax

The syntax for declaring a mapping is

mapping(keyType => valueType) <access specifier> <name>;

where keyType represents the type of the key and valueType represents the type of the associated value. The access specifier determines the visibility of the mapping, such as public, private, or others.

Example:

```
//SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;



contract Mappings{

   mapping(address => string) public AddrToName;

}
```

In the above example, ‘keyType’ is taken as **address,** ‘valueType’ is taken as **string**, and the **mapping** is named as ‘AddrToName’. By using the above **mapping**, we can access a **string** value using an **address** data type value.

Exercise

Make a contract named ‘Mapper’. In it make a **mapping**, that has ‘keyType’ **string** (which will represent name of a Wallet) and ‘valueType’ **uint** (which will represent the balance of the account). Name the **mapping** as NameToBalance.

1.2 Adding Values To A Mapping

When adding values to a mapping, you can assign a value to a specific key using the syntax

_mapping[key] = value;. This allows you to store and update data in the mapping.

Example:

```
function AddValue(address _addr, string memory _name) public{


      AddrToName[_addr] = _name;

   }
```

Exercise

In the contract named ‘Mapper’. Make a function named ‘set’ that assigns a value to the **mapping.**

1.3 Retrieving Value From A Mapping

To retrieve values from a mapping, you can use the mapping's name followed by the desired key, like **_value = _mapping[key];**. This allows you to access the associated value using the corresponding key.

Example:

```
function get(address _addr) public view returns(string memory){

       return AddrToName[_addr];

   }
```

The above function returns the **string** value associated with the particular **address** value that the user has entered. If a key has not been assigned a value, it will have an initial value of 0, as

‘Solidity treats unused space as 0.’

Exercise

In the contract named ‘Mapper’. Make a function named ‘get’ that gives you the value of a **mapping.**

1.4 Deleting A Mapping

You can also remove a mapping by using the **delete** keyword in Solidity. This removes the value assigned to the key and converts it to the initial value ‘0’.

Example:

```
function remove(address _addr) public{

       delete AddrToName[_addr];

   }
```

If the given input **address** has been assigned a **string** value previously, then the above function will reset that value to ‘0’.

Exercise

In the contract named ‘Mapper’. Make a function named ‘reset’ that resets the value of a **mapping.**

1.5 Nested Mapping

Solidity also allows ‘Nested Mapping’ as well. Thanks to it you can assign multiple keys to a single value. The syntax for ‘nested mapping’ is given as:

mapping(keyType => mapping(keyType => valueType)) <access specifier> <name>;

Example:

Let us modify the above example such that it also includes name of the wallet as a key as well.

```
//SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

contract Mappings{

   mapping(address => mapping(string => string)) public AddrWalletToName;


   function AddValue(address _addr, string memory _walletname, string memory _name) public{

      AddrWalletToName[_addr][_walletname] = _name;

   }

    function get(address _addr,  string memory _walletname) public view returns(string memory){

      return AddrWalletToName[_addr][_walletname];

   }

   function remove(address _addr,  string memory _walletname) public{

      delete AddrWalletToName[_addr][_walletname];

   }                                                                                    }
```

Exercise:

a) Observe and deploy the above contract.

b) Modify the above contract (mapping and all functions) such that the ‘name’ of **string** type is also a key, and the **mapping** gives a value of **uint** type named ‘balance’ (which will represent the current balance of the person).

Hint: `mapping(address => mapping(string => mapping(string => uint))) public AddrWalletToName;`

1.6 Rules and Principles

It's important to note some rules and principles when working with mappings.

- The **key** type must be selected from the default types in Solidity, such as uint or address, while the **value** type can be any custom type.
- Mappings should be declared as **storage** variables or state variables and cannot be used as function arguments or return results in public functions.
- If a mapping is declared as public, Solidity automatically generates a **getter** function to retrieve the value by the key.
- In certain cases, you may need to use **nested mappings** to handle multiple relationships. Nested mappings allow you to create a mapping within another mapping, providing a way to manage complex data structures.

Mappings are a powerful tool in Solidity for organizing and retrieving data efficiently. By understanding the syntax, principles, and best practices associated with mappings, you can effectively store and retrieve key-value pairs in your Solidity contracts.

Exercise

a) Make a contract named ‘NameMap’. In it, define a **mapping**, having keys ‘YearOfBirth’ of **uint** type, ‘WalletAddress’ of **address** type, and ‘Codename’ of **string** type. The **mapping** should give a value named ‘UserName’ of **string** type. Name the **mapping** as ‘UserMap’.

b) Create a setter function for the above **mapping** named ‘setMap’.

Hint:
```
function setMap(uint _yearofbirth, address _wallet, string memory _codename, string memory _username) public
```

c) Create a getter function for the above **mapping** named ‘getMap’.

Hint:
```
function getMap(uint _yearofbirth, address _wallet, string memory _codename) public view returns(string memory _username)
```

d) Create a ‘reset’ function that **deletes** the value of a particular set of keys.

Hint:
```
function reset(uint _yearofbirth, address _wallet, string memory _codename) public
```

2 Data Locations

In Solidity, variables can be declared with different data locations to explicitly specify where the data is stored. The three main data locations are storage, memory, and calldata.**Storage:**

- Variables declared as storage are state variables that are **stored on the blockchain**.
- These (state) variables are persistent and can be accessed from anywhere inside the smart contract.
- Global variables are by default storage variables.
- It's important to note that writing to storage variables incurs gas fees since it involves modifying the blockchain state.

NOTE:

1) When ‘storage’ (a state variable of the contract) is assigned to the ‘local storage’ (in a function), a **reference** will be created, and changing value of the new variable will affect the original one.

2) Assigning ‘storage’ to ‘memory’ creates **independent** copies, and changes to one will not affect the other; and vice versa.

EXERCISE:

1) Observe and deploy the following smart contract.
2) Get the initial array elements using **get()** function, transact the **fStorage()** function and re-call the **get()** function.
3) Change ‘storage’ to ‘memory’ in `uint[] storage xStorage = arr;` ,and observe the changes.

```
  // SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract StorageDLoc {
  uint[] arr = [1,2,3];                // state variable

    function fStorage() public{
        // Declare a local storage variable xStorage
        uint[] storage xStorage = arr;
        xStorage[0] = 100;
    }


    function get() public view returns (uint[] memory){
        return arr;
    }
}
```

1) **Memory:**

- Variables stored in memory are **temporary** and exist only during the execution of a function.
- They are not written to the blockchain and are mainly used for calculations or operations within the function's scope.
- Once the function completes, memory variables are destroyed.
- They are not accessible outside the function.

NOTE:

Assigning ‘memory’ to ‘memory’ will create a **reference**, and changing the new variable will affect the original variable.

EXERCISE:

1) Observe and deploy the following smart contract.
2) Does **modifyMemory()** function create copy or reference?

```
  // SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

 contract MemoryAssignment {

    function modifyMemory(uint[] memory originalArray) public pure
returns (uint[] memory) {
        // Assign the input array to a new variable in memory
        uint[] memory newArray = originalArray;

        // Modify a value within the new array
        if (newArray.length > 0) {
            newArray[0] = 999;
        }

        return originalArray;  // Return the original array
    }
}
```

2) Calldata:

- Calldata is a special data location that is **valid only for parameters** of external contract functions.
- It acts similarly to memory and is a **non-modifiable** and **non-persistent** area where function arguments are stored.
- Calldata variables are used to pass data into a function, and the data passed is not modified.
- This data location is particularly useful when interacting with **external** contracts.

NOTE:

Calldata variables cannot be modified and are read-only.

EXERCISE:

Observe and deploy the following smart contract.

Try to modify array **_x** by uncommenting `_x[0] = 0;`, and observe what happens.

```
  // SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

 contract calldataDloc {

    function fCalldata(uint[] calldata _x) public pure
returns(uint[] calldata){

        // _x[0] = 0;

        return(_x);

    }

}
```

Choosing the appropriate data location is crucial for optimizing gas usage and ensuring the correct behavior of your smart contracts.

By explicitly specifying the data location, you can control where the data is stored and accessed, whether it's

➔ on the blockchain (storage)
➔ within a function (memory),
➔ as persistent function arguments (calldata).

EXERCISE:

1. Create a struct **‘CAR’** with the attributes of Make, model and Variant.

2. Create a function **initCar1()** functions that creats a storage struct **reference** within a function.

3. Create a function **initCar2()** functions that creats a memory struct **copy** within a function.

```
struct CAR {...}


CAR car("Toyota", "Prado", 2000);


function initCar1() public returns (Car state, Car local) {

     CAR storage _car = car;

    ...

}

function initCar2() public view returns (Car state, Car local) {

     CAR memory _car = car;

    ...

}
```