LESSON SC 2 – Data Structures

University of West Attica
Department of Electrical and Electronics Engineering
Ioannis Christidis
Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

This is the first step for our project **Instant Car Renting (ICR)**

In this lesson we will learn about *data structures* in solidity:

- `Structs`
- `Arrays`
- `Mappings`

Basics

Start by creating a new file in your `sc-iot-course` *folder and name it* `ICR.sol`.

In your new file write the basics of a SC

1. *License (MIT)*
2. *Solidity Version (0.8.26)*
3. *Name of the SC (ICR)*

It is generally a good practice to *name the SC as you name your file*.

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

pragma solidity 0.8.26;

contract ICR {

}
```

Structs

A `struct` in Solidity is a *data type that allows* you *to group multiple related variables into a single entity*. It's like creating a custom "object" with specific properties

`structs` are useful for *representing complex data types*, such as user profiles, orders, or transactions, which have multiple attributes.

```
struct Car {
    address mc; // microcontroller address
    uint256 price;
    bool occupied;
}
```

So, for our project we will create a `struct` named `Car` that will contain the car's properties.

Each car will have:

1. *A microcontroller with a blockchain account (in the* `struct` *we will add the account's* `address`*)*
2. *A price to rent the car* which will be a `uint256`
3. A `bool` that is `true` when the car is rented or occupied and `false` when it is not.

Arrays

`arrays` are *collections of elements of the same data type*, and they can be *fixed-size* or *dynamic* (similar to `data`).

```
    uint[5] public fixedArray; // A fixed-size array of 5 elements
    uint[] public dynamicArray; // A dynamic array with no predefined size
```

You can ***declare an array for every data type like the*** **`Car struct`** we just created.

We will create a `dynamic array` since we do not know the exact number of cars that will be added to our .

```
    Car[] public carList; // A dynamic array of cars
```

Currently, our `array` does not have any entries, but we will add some in the future, so to get the ***size of the*** **`array`** we can use:

```
   array.length
```

Let's create a `getter function` that *returns the length of our* `carList`:

```
    function getCarListLength() public view returns (uint256) {
        return carList.length;
    }
```

Arrays (2)

To *add a new element on an* `array` we can use:

```
array.push(element);
```

Now that we know that we can create a *setter function that adds a new* `Car` *to our* `carList`.

We will call it `registerCarToList` and we will pass the car's information as parameters, the `address` of the microcontroller (`_mc`) and its price. We will make the car unoccupied when it is registered so we will not add an extra parameter and we will hardcoded it as `false`.

Since we are adding an element to an `array` we are *changing the state* so this function will make a transaction when it is used.

```
function registerCarToList(address _mc, uint256 _price) public {
    Car memory car = Car(_mc, _price, false);
    carList.push(car);
}
```

In the function you see above we first create a temporary `Car` with its properties (`_mc, _price, false`) and we name it `car`. It is temporary because we used the `memory` keyword which means that it is *saved in a temporary data storage only for the duration of the function*.

We then add `car` to `carList` by using the `push` method saving permenately in the array.

Arrays (3)

To *retrieve an item from an* `array` *we can pass its position in the array*:

```
array[itemPosition];
```

To view a `Car` from our `carList` we will create a *getter function*. This function will take as *input the position of the* `Car` *in the array (or ID) and return it*.

```
    function getCarFromList(uint _carId) public view returns (Car memory) {
        return carList[_carId];
    }
```

Notice that the `Car` to be returned is also in `memory` so, it is temporary.

The returned result will be the `Car` with `_carId` position in the `carList`.

Now, we will do the same, again, but this time we will be using a unique data structure in solidity called `mapping` and then we will compare `arrays` and `mappings` to choose the best for our scenario.

Mappings

`mappings` are *key-value stores that enable efficient and direct access to data*. They are similar to hash tables or dictionaries in other programming languages.

```
mapping(KeyType => ValueType) public mappingName;
// example map that pairs an id to a name
mapping(uint id => string name) public names;
```

We will create a `mapping` that pairs an ID to a `car`, similar to positions in arrays.

```
mapping (uint256 _carId => Car _car) public carMap;
```

`mappings`*do not have a length method*, like `arrays` do, so *to keep track of entries we will create a `uint` that will give us the ID of the next card to be added, simulating arrays' length*.

```
uint256 public nextCarId;
```

In a`mapping`*we can add any key-value* we want. For the `names` example above we could add as `ID` the number 7 and as `name` Bob and *skip all the previous ones.*

Also, if we try to *get the value by using a key that does not have a value yet* we will still get a result, but *it will be the default value of that data type*. For example, in the names mapping if we add the key `105` it will return an empty `string`.

In our project however we need to follow the order of entries, which means that the first car will have `ID = 0` the second will have `ID = 1` e.t.c.

Mappings (2)

To add a `Car` in our `mapping`, we will create a *setter function* that takes the same parameters as the ones we used in the `array` example.

```
function registerCarToMapping(address _mc, uint256 _price) public {
    Car memory car = Car(_mc, _price, false);
    uint256 currentCarId = nextCarId;
    carMap[currentCarId] = car; // maps car to currentCarId
    nextCarId++;
}
```

Similar to what we did with the arrays we first create a `Car` named `car`

Next, we declare a `uint256` called `currentCarId` and we make it equal to `nextCarId` which as a reminder is the uint that tracks the ID of the next car to be added.

We then map the `currentCarId` with the `car`.

Finally, we increase the `nextCarId` by one getting it ready for the next time we want to register a car.

Next, similar to `arrays` we will also create a *getter function* to retrieve a car by its ID.

```
function getCarFromMap(uint _carId) public view returns (Car memory) {
    return carMap[_carId];
}
```

Comparison of `Arrays` and `Mappings`

Feature	`Arrays`	`Mappings`
Definition	Ordered collection of elements of the same type.	Key-value pairs for direct lookups.
Access	By Index	By key
Default Value	Throws an error for out-of-bounds indices.	Returns default value of the value type for missing keys.
Iteration	Possible	Requires tracking of keys (e.g. nextCarId)
Order preservation	Elements are stored in the order they are added.	Require custom functionality
Gas Cost for Access	**Efficient for sequential data and iteration.**	**Efficient for random key lookups.**
Gas Cost of Growing	**Costly for large arrays due to copying and resizing.**	**Keys are added dynamically without reallocation.**

We will go with the *most cost-efficient approach* and since we do not plan to do iterations we will continue with `mappings`.

On the next slide you will find the full code only with mapping.

Coding Completed

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

contract ICR {

    struct Car {
        address mc;
        uint256 price;
        bool occupied;
    }

    mapping (uint256 _carId => Car _car) public cars;

    uint256 public nextCarId;

    function registerCar(address _mc, uint256 _price) public {
        Car memory car = Car(_mc, _price, false);
        uint256 currentCarId = nextCarId;
        cars[currentCarId] = car;
        nextCarId++;
    }

    function getNextCarId() public view returns (uint256) {
        return nextCarId;
    }

    function getCar(uint _carId) public view returns (Car memory) {
        return cars[_carId];
    }

}
```

Outro

Great job, we created the basis of or SC.

In the next lesson we will go over conditional logic and iteration.