BCDV 1010
Smart Contract Development Essentials

2023 January

week 01 - class 04

Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.
Value Types:

The following types are called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

1. Booleans
2. Integers
3. Address
4. Strings
5. Functions
6. Enums

Refrence Types:

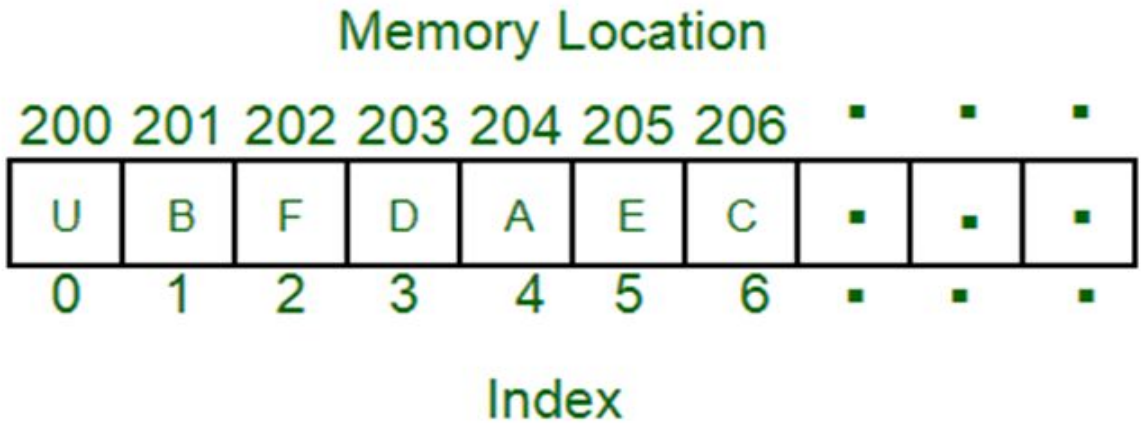
Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used.

1. Arrays
2. Structs
3. Iterable Mappings

Arrays

What is an Array?

An array is a collection of items of the same data type stored at contiguous memory locations.



Types of Array in Solidity

- Fixed-size byte arrays

  The type of an array of fixed size `k` and element type `T` is written as `T[k]`

  `E.g: An array of the addresses`

```
address[12] registry;
```

To retrieve the value:

```
function retrieve() public view returns (address){
    return registry[3];
}
```

Types of Array in Solidity

- Dynamically-sized byte array

  The type of an array of dynamic size as T[]

  For example, an array of 5 dynamic arrays of uint is written as uint[][5]. The notation is reversed compared to some other languages. In Solidity, X[3] is always an array containing three elements of type X, even if X is itself an array. This is not the case in other languages such as C.

```
address[] dynamic_registry;
```

```
function retrieve_dynamic(uint _index) public view  returns (address){
    if(_index < dynamic_registry.length) {
        return dynamic_registry[_index];

    }else{
        revert("invalid Index");
    }

}
```

Types of Array in Solidity
(Dynamically-sized byte array)

Memory arrays with dynamic length can be created using the `new` operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the `.push` member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

```
function f(uint len) public pure {
    uint[] memory a = new uint[](7);
    bytes memory b = new bytes(len);
    assert(a.length == 7);
    assert(b.length == len);
    a[6] = 8;
}
```

Array Initialization:

1. Arrays can be initialized statically as follows:

```
uint256[3] balances = [1, 2, 3];
```

2. If Initializing inside a function:

```
function f(uint len) public pure {
    uint[] memory a = new uint[](7);
    bytes memory b = new bytes(len);
    assert(a.length == 7);
    assert(b.length == len);
    a[6] = 8;
}
```

Array Members

length:
Arrays have a length member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

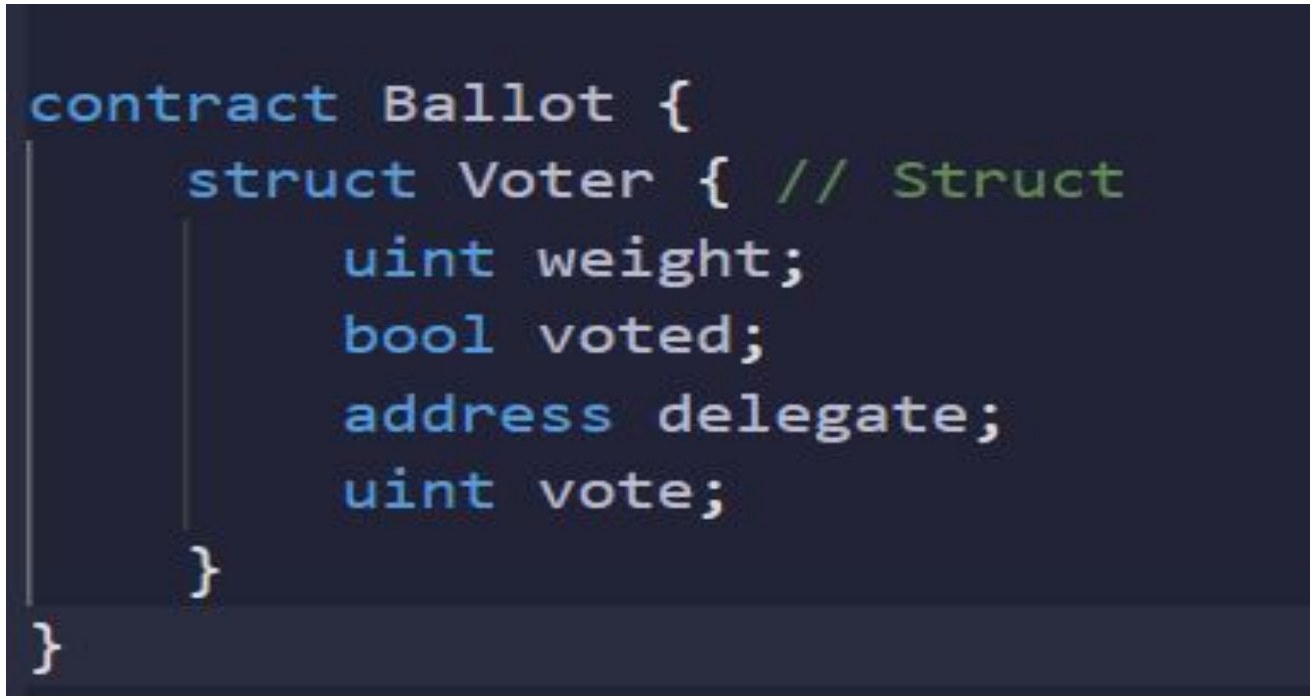
push():
Dynamic storage arrays and bytes (not string) have a member function called push() that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like x.push().t = 2 or x.push() = b.

push(x):
Dynamic storage arrays and bytes (not string) have a member function called push(x) that you can use to append a given element at the end of the array. The function returns nothing.

pop():
Dynamic storage arrays and bytes (not string) have a member function called pop() that you can use to remove an element from the end of the array. This also implicitly calls delete on the removed element. The function returns nothing.

Structures

Structs are custom-defined types that can group several variables.
e.g:



Why do we need Structs?
If we have to group particular values of different types,
Structs enable us to do so.
E.g. In the case of geolocation, we always need to store the Longitude and Latitude together.
{ lat: -34.397, lng: 150.644 }

```
struct Coordinate {
    int256 lat;
    int256 lng;
}
```

Initializing new Struct

The struct variable can be initialized by passing it JSON format values:

```
contract Ballot {
    struct Voter { // Struct
        uint age;
        bool voted;
        address delegate;
        uint vote;
    }

    Voter public voter;

    constructor(){
    voter=Voter({
        age: 37,
        voted: false,
        delegate: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db,
        vote: 1}
     );
    }
}
```

Initializing new Struct

If Initializing the struct inside the function, the storage/memory type of the variable needs to be specified:

```
function contribute() public payable {
    Voter memory v = Voter({
    age: 37,
    voted: false,
    delegate: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db,
    vote: 1}
);

}
```

Retrieving new Struct

While retrieving the struct it needs to retrieve in a memory object:

```
function getVoter() public view returns(Voter memory){
    return voter;
}
```

Modifying the Struct

The structs can be easily modified by accessing the keys inside it.

```
function addVote(uint count) public returns (uint){
    voter.vote+=count;
    return voter.vote;
}
```

Enums

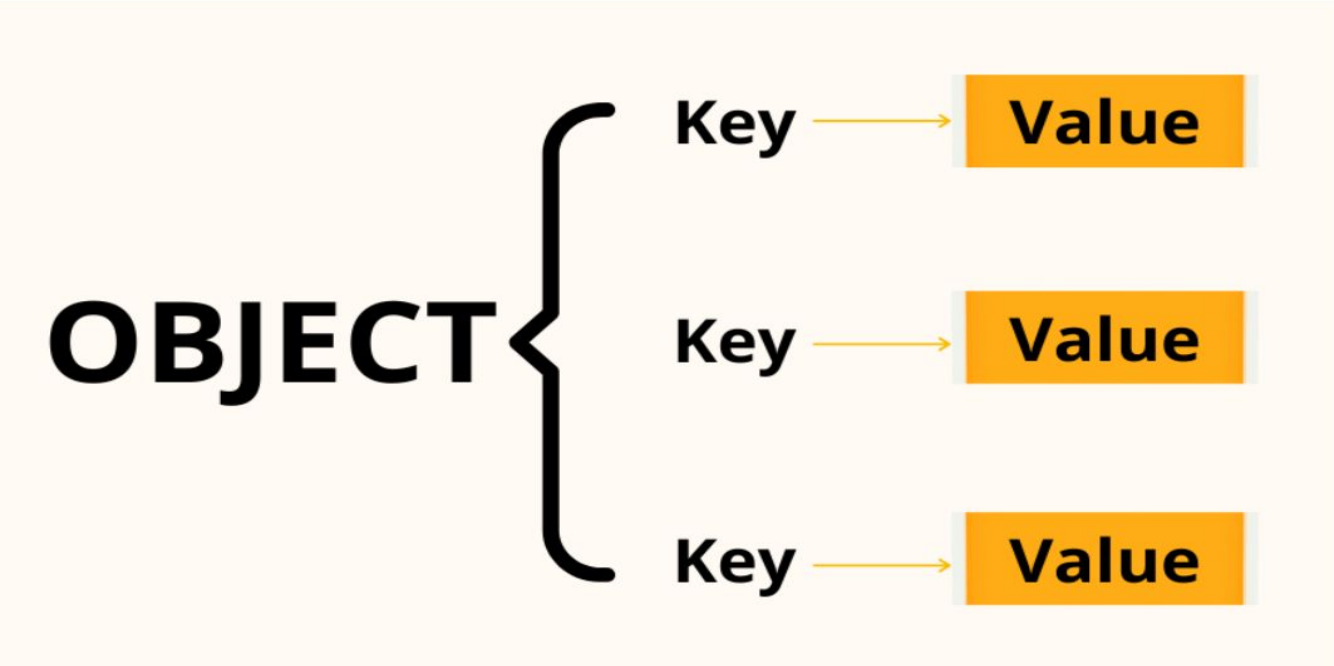
Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a Panic Error otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

```
pragma solidity ^0.8.8;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(ActionChoices.GoLeft);
    }

    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

Mappings

What are key-value pairs?

A key-value pair **consists of two related data elements: A key, which is a constant that defines the data set and a value, which is a variable that belongs to the set**.



Mapping types use syntax `mapping(KeyType => ValueType)` and variables of mapping type are declared using the syntax `mapping(KeyType => ValueType) VariableName`. The `KeyType` can be any built-in value type, `bytes`, `string`, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. `ValueType` can be any type, including mappings, arrays and structs.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount, "ERC20: Not enough funds.");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

Iterable Mappings

With the mappings, you can easily access the value for the particular key, but accessing the list of keys in the mapping is not possible by itself. For that, we can implement Iterable Mappings where we can also access the keys in the mappings.

Please refer here for more information on Mappings:

References

Memory Layouts:

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

- `0x00` - `0x3f` (64 bytes): scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory **pointer**)
- `0x60` - `0x7f` (32 bytes): zero slot

Scratch space can be used between statements (i.e. within the inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory **pointer** points to `0x80` initially).

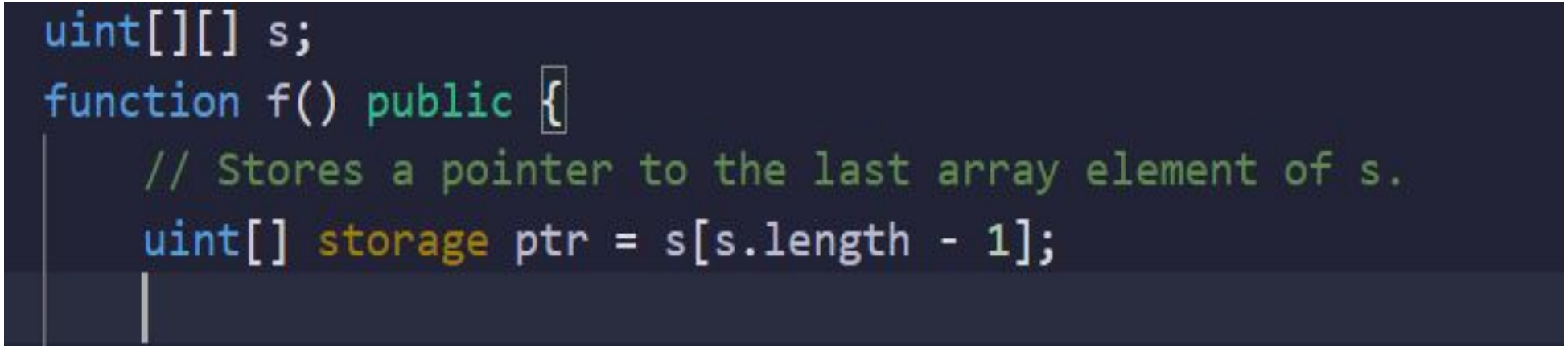
Solidity always places new objects at the free memory **pointer** and memory is never freed (this might change in the future).

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `bytes1[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are **pointer**s to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

Shallow copy

A shallow copy of an object is **a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made**.

In the example below we are referencing a pointer to an array which is already initialized



Deep copy

A deep copy of an object is **a copy whose properties do not share the same references (point to the same underlying values) as those of the source object from which the copy was made**.

In the example below, we have initialized a new variable and pushed element ‘3’ in it. This array has its own new location in the memory.

```
// Initialize a new Array with the size 7
uint[] memory new_var=new uint[](7);

// Inserting a new element '3' in the array
new_var[new_var.length -1]= 3;
```

Reference links

https://docs.soliditylang.org/en/v0.8.17/types.html#value-types

https://docs.soliditylang.org/en/v0.8.17/types.html#arrays

https://docs.soliditylang.org/en/v0.8.17/types.html#structs

https://docs.soliditylang.org/en/v0.8.17/types.html?highlight=enums#enums

https://docs.soliditylang.org/en/v0.8.17/types.html?highlight=enums#mapping-types

https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_memory.html?highlight=pointer