LAB 31-07-2023

1 Functions

1.1 Introduction

- In Solidity, functions play a crucial role in organizing code, improving efficiency, and promoting code reuse.
- They allow developers to **encapsulate** a group of code that can be used repeatedly throughout the program, reducing the need for repetitive code and optimizing runtime performance.
- By dividing a program into smaller, manageable pieces of code, functions enhance code **readability and maintainability.**

1.2 Declaring a Function

When declaring a function in Solidity, the **'function'** keyword is used, followed by a unique name **that doesn't conflict with reserved keywords**. Functions can also include a list of parameters, specifying their names and data types.

While the return value of a function is optional, Solidity requires defining the return type at the time of declaration.

1.3 Calling a Function

To execute a function in Solidity, it is simply invoked by writing its name at the desired location. Parameters can be passed to functions during the function call, with multiple parameters separated by commas.

Solidity allows for the use of return statements, which are optional but powerful. Placed as the last statement within a function, return statements are used to return values from the function.

Solidity supports the return of multiple values from a single function, as long as the data types of the return values are defined in the function's declaration.

1.4 Syntax:

Solidity has 'function' syntax as following:

```
function <func name>(<parameter types>) [public|private|internal|external]
[pure|view|payable] [returns (<return types>)]
```

1.5 View and Pure Functions

1.5.1 Introduction

- Solidity introduced "pure" and "view" keywords to optimize gas fees associated with smart contracts.
- By using these function types, you can avoid paying gas for accessing or modifying state variables stored on the blockchain.
- "Pure" functions ensure no state variables are accessed or modified, while "view" functions allow reading without modification.
- This optimization strategy reduces gas costs, making contract interactions more efficient and cost-effective on the Ethereum blockchain.

Understanding the distinction between these two types of functions is essential for writing secure and efficient smart contracts on the Ethereum blockchain.

1.5.2 View Functions:

➔ View functions in Solidity are designated as read-only functions.

➔ They offer a guarantee that the state variables within a contract will not be modified when called.

➔ Think of them as functions that allow you to "look" at the state variables without making any changes to them, much like observing a snapshot of the current state.

Here are some key characteristics and guidelines regarding view functions:

1) **Read-Only:** View functions cannot modify any state variables within the contract. Their purpose is solely to retrieve and provide information from the blockchain.
2) **Compiler Warnings**: If a view function attempts to modify state variables, emit events, create other contracts, use self-destruct, transfer ethers via calls, or make non-view/pure function calls, the compiler will throw a warning. It serves as a safety mechanism to ensure adherence to the intended read-only nature of view functions.
3) **Default for Get Methods:** By default, the functions generated by Solidity for retrieving state variables (known as "get" methods) are view functions.

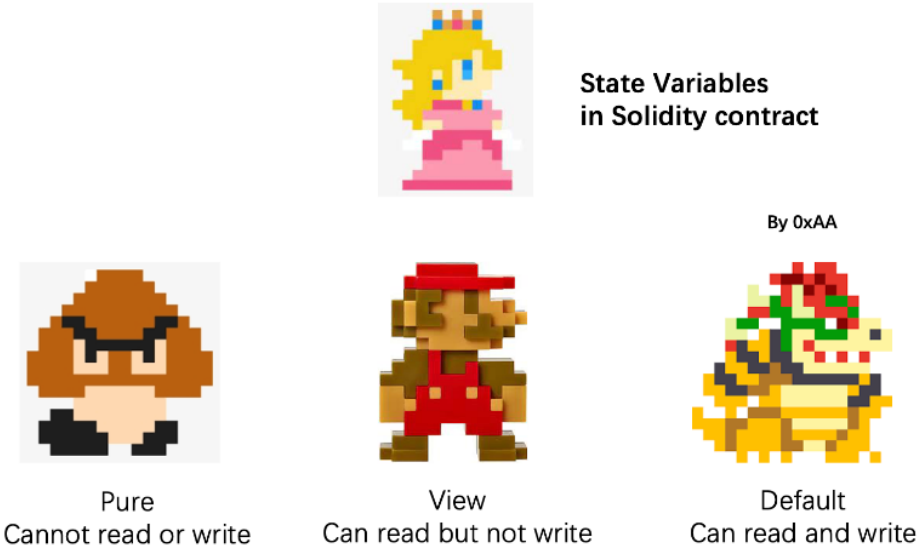
1.5.3 Pure Functions:

- ➔ Pure functions in Solidity go even further than view functions by disallowing access to state variables altogether.
- ➔ They solely rely on the values passed as parameters or local variables within the function and return computed results based on them.

Here are some key characteristics and guidelines regarding pure functions:

1) **No Access to State Variables:** Pure functions cannot read or modify any state variables. Their purpose is to perform computations or transformations using only the provided inputs.
2) **Compiler Warnings:** If a pure function attempts to read state variables, access addresses or balances, use global variables like block or msg, or make calls to non-pure functions, the compiler will issue a warning. This ensures that pure functions maintain their isolation from the contract's state.
3) **Return Values Only:** Pure functions are designed to return values derived from the given input parameters, making them suitable for calculations or data transformations that do not require access to the contract's state.

Understanding the distinction between view and pure functions is crucial for writing secure and efficient smart contracts in Solidity.



1.6 Further principles relating to functions:

1) Functions can return multiple values.

```
function returnMany() public pure returns (uint, bool, uint) {
      return (1, true, 2);
  }
```

2) Return values can be named.

```
function named() public pure returns (uint x, bool b, uint y) {
      return (1, true, 2);
  }
```

3) Return values can be assigned to their name.

```
function assigned() public pure returns (uint x, bool b, uint y) {
      x = 1;
      b = true;
      y = 2;
  }
```

4) Use destructuring assignment when calling another function that returns multiple values.

```
  function destructuringAssignments() public pure
      returns (uint, bool, uint, uint, uint)
  {
      (uint i, bool b, uint j) = returnMany();

      // Values can be left out.
      (uint x, , uint y) = (4, 5, 6);

      return (i, b, j, x, y);
  }
```

5) Call function with key-value inputs.

```
  contract Prototype {
    function someFuncWithManyInputs(
        uint x,
        address a,
        bool b,
        string memory s
    ) public pure {

    }

    function callFuncWithKeyValue() public pure  {
        return
            someFuncWithManyInputs({a: address(0), b: true, s: "Function
Call with Key Value", x: 1});
    }
}
```

EXERCISE

Deploy and observe the following contract.

```
 // SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
 contract EncapFuncCalls {
    function someFuncWithManyInputs(
        uint x,
        uint y,
        uint z,
        address a,
        bool b,
        string memory c
    ) public pure returns (uint _x,uint _y,uint _z,address _a,bool
_b,string memory _c ) {
        _x = x;
        _y = y;
        _z = z;
        _a = a;
        _b = b;
        _c = c;
    }

    function callFunc() external pure returns (uint _x,uint _y,uint
_z,address _a,bool _b,string memory _c ) {
        return someFuncWithManyInputs(1, 2, 3, address(0), true, "Simple
Function Call");
    }

    function callFuncWithKeyValue() external pure returns (uint _x,uint
_y,uint _z,address _a,bool _b,string memory _c ) {
        return
            someFuncWithManyInputs({a: address(0), b: true, c: "Function
Call with Key Value", x: 1, y: 2, z: 3});
    }
}
```

EXERCISE

Create three separate functions that return your Name, Gender and MetaMask's wallet address in three different forms as introduced in **Section 1.6 point 1,2 and 3.**

EXERCISE

Create a Solidity function that takes radius **'r'** as input and returns the circumference of the circle.

Hint:

Circumference is found by 2*pi*r, where pi is 3.1415

EXERCISE

1) There are some errors in the following code, correct them.
2) Deploy and observe the updated version of code.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract FunctionTypes{
    uint public number = 4;

    // default
    function add() public{
        number += 2;
    }
    // pure
    function addPure() public pure {
         number += 2;
    }
    // view
    function addView() public view  {
         number += 2;
    }
    }
```

Hint:

1.
```
function addPure(uint256 _number) public pure returns (uint temp_number)
```

2.
```
function addView() public view returns (uint temp_number)
```

3. **number += 2** is here, equivalent to **number = number + 2.**

2 Arrays

Arrays are a fundamental data structure in Solidity used to store a collection of values of the **same type**, such as integers, addresses, or bytes.

In Solidity, there are two types of arrays

2.1 Fixed-sized arrays

- These have a **predetermined** length specified at the time of declaration.
- They are declared using the format T[k], where T represents the element type and k represents the length of the array.
- It should be noted however that indexing of an array starts at **zero** and ends at **'k-1'**
- By default, array values will be the default values of the given data type, if they have not been already defined.

Example:

```
uint[8] array1;
byte[5] array2;
address[100] array3;

int[2] array4;
```

2.2 Dynamically-sized arrays

- Do not have a fixed length and can grow or shrink as needed.
- They are declared using the format **T[]**, where T represents the element type.
- Additionally, there is a special case for the bytes type, which is also a dynamically-sized array but can be declared simply as **bytes without the need for []**.

Example:

```
uint[] array4;
bytes[] array5;
address[] array6;
bytes array7;
```

2.3 Built-in Functions:

Arrays have some built-in members and functions.

1. Length:

- The **length** member provides the number of elements in an array.
- It can be used with both fix-sized and dynamic arrays.
- It is of 'uint' type and can never be negative.

```
uint[3] public arr1;
    function getlength() public view returns(uint){
        return arr1.length;
    }
```

EXERCISE

1) Deploy the following contract
2) Observe the output of **getlength()** function and also explain the reason.

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

contract DArray {
       uint[] public arr;
       function getlength() public view returns(uint){
            return arr.length;
    }
}
```

2. push():

- Dynamic arrays also have a **push()** function that adds a new element of value x to the end of the array, allowing the array to expand.
- It increases the length of the array by 1.
- It can only be **used with dynamic arrays**, as it changes the length of the array.

```
    int[] public arr;

    function upush(int _x) public{
        arr.push(_x);
    }
```

3. pop ():

- Dynamic arrays have a **pop()** member that removes the last element of the array.
- It decreases the length of the array by 1.
- It removes only the last value of an array.
- It can only be used with dynamic arrays.
- You will get an error if you try to pop an already empty array.

```
function upop() public{
        arr.pop();
    }
```

4. delete:

- It replaces the value of an array with its default value.
- It does not affect the length of an array.
- It can be used with both fix-sized arrays and dynamic arrays.

```
function udelete(uint _index) public{
        delete arr6[_index];
    }
```

- You can also delete an entire array.

```
function udelete1() public{
        delete arr6;
    }
```

- For dynamic array, deleting it entirely will convert it into an empty array.
- For fix-sized array, deleting it entirely will convert all its elements to their default values.

2.5 Returning An Array

- You can return an entire array by using a getter function.
- **'memory'** keyword or **'calldata'** keyword must be used in return type for an array.

1) For a **fixed-sized array,** you must also mention the length of the array as well.

```
uint[2] public arr1 = [5, 6];

    function get1() public view returns(uint[2] memory){
```

```
        return arr1;
    }
```

2) For **dynamic arrays**, length of the array will not be mentioned.

```
uint[] public arr2;

    function get2() public view returns(uint[] memory){
        return arr2;
    }
```

2.6 Returning & Giving Value To Element Of An Array

- For returning/giving value to a single element of an array, you will take the index of the element to be returned as uint type input. In the example below, we will take _i as our index.

```
//Returning Element of an array
    uint[3] public arr3 = [2, 6, 9];
    function getElement(uint _i) public view returns(uint){

        return arr3[_i];
    }
```

```
//Giving value to an element of an array
    uint[5] public arr4;
    function give(uint _i, uint value) public{

        arr4[_i] = value;
    }
```

EXERCISE

a) Make a uint type fix-sized array of length 4 named arr1.
b) Convert the 0^{th} and 3^{rd} element of arr1 into 2 and 10 respectively.
c) Make a getter function for arr1.

EXERCISE

a) Declare a dynamic array of type string named arr2.
b) Using push(), fill the array with your complete name.
c) Return the length of the arr2 array.

d) Using pop() remove the last name.

e) Make a getter function for arr2, that now returns your first name.

EXERCISE

There are 4 errors in the code given below, correct them and then run the updated version of the code.

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

contract Array{
    int[4] public array;

    function ipush() public{
        array.push(-5);
        array.push(0);
        array.push(5, 10);
    }

    function get() public view returns(int[]){
        return array;
    }

    function getlength() public view returns(int){
        return array.length;
    }
}
```

Hint:

- push() and pop() cannot be used with fixed-sized arrays.
- You can push only one value at a time.
- When returning an entire array, you need to use ‘memory’ or ‘calldata’ keyword.
- Array length cannot be negative.

EXERCISE

Create a Solidity contract that stores a dynamic array of integers. Implement functions to

1. add elements to the array

```
myarray.push(_number);
```

2. retrieve elements at specific indices

```
return myarray[_index];
```

3. delete elements at specific indices

```
delete myarray[_index];
```

4. pop element from array.

```
myarray.pop();
```