LAB 04 August, 2023

Essential in Solidity, loops are foundational code structures enabling the repetitive execution of code. They prove invaluable for tasks requiring **multiple iterations** or when navigating through data collections.

Solidity accommodates various loop types:

1 While Loop

- The '**while**' loop stands as a foundational loop in Solidity, serving to iteratively execute a designated block of code contingent upon a specified condition's validity.
- It is commonly used when the number of iterations is unknown beforehand.

Syntax:

```
while (condition) {
    // Code to be executed
}
```

- Here **'while'** is a keyword followed by condition in **'( )'**. The piece of code needs to be repeated as long as condition is satisfied, comes in the **'{ }'.**
- The loop first checks the condition. If it evaluates to true, the code inside the loop is executed.
- After each iteration, the condition is checked again until it becomes false, at which point the loop terminates

Example:

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

contract Loop{
    uint[] public arr;

    function NaturalNum(uint max) public returns(uint[] memory){
        uint i = 1;
        while(i <= max){

            arr.push(i);
            i++;
        }
        return arr;      }
   }
```

EXERCISE:

1) Run and deploy the above smart contract.
2) Modify the above smart contract such that it outputs the multiple of **‘4’**, until the ‘**max**’ number

Hint:

1.
```
uint i = 4;
```

2.
```
  i+=4;
```

Example:

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

pragma solidity ^0.8.17;

contract NumberProcessor {

    function calculateSumAndAverage(uint256[5] memory numbers) public pure
returns (uint256 sum) {

        uint256 totalSum = 0;

        uint256 i = 0;

        while(i < numbers.length) {

            totalSum += numbers[i];

            i++;

        }

        sum = totalSum;

    }

}
```

EXERCISE:

1) Run and deploy the above smart contract.
2) Modify the above smart contract such that it also outputs the average of entered array of numbers. **(average = totalSum / numbers.length;)**

EXERCISE:

Complete the following smart contract to print **Fibonacci series** of n terms where n is input by user :

Fibonacci series: 0 1 1 2 3 5 8 13 24 .....

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

pragma solidity ^0.8.17;

contract FibonacciSeries {

    function generateFibonacci(uint256 n) public pure returns (uint256[]
memory) {

        uint256[] memory fibonacciSeries = new uint256[](n);

        if (n >= 1) {

            fibonacciSeries[0] = 0;         }

        if (n >= 2) {

            fibonacciSeries[1] = 1;         }

/********************** Change this Part Only ********************************

1. Initialization: unit i = 2.

2. Condition :      i < n

3. Formula:         fibonacciSeries[i] = fibonacciSeries[i - 1] +
fibonacciSeries[i - 2];

4. Increment :      i++

******************************************************************************
*/

        return fibonacciSeries;             }

}
```

2 Do-While Loop

- The do-while loop is similar to the while loop, but it guarantees that the code block is executed at least once, even if the condition is initially false.
- The condition is checked at the end of the loop.

Syntax:

```
do {
    // Code to be executed
} while (condition);
```

The **do** keyword is written first, followed by the curly brackets **{ }**, in which the loop body to be executed is enclosed. After the curly brackets the **while** keyword is written followed by the condition of the loop enclosed in circular brackets ( ).

2.1 Explanation

In Do-While loops, code block is executed first, and then the condition is evaluated. If the condition is true, the loop continues to the next iteration. If the condition is false, the loop terminates.

Example

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

contract Loop{

    //The value of cond that you will input will decide
    //the number of times the loop is run
    function iterate(uint cond) public pure returns(uint){

        uint i;
        uint iteration;

        do {
            iteration ++;
            i++;
        } while(i < cond);

    //If you input the value of 'cond' as 0, the condition will be false
    //on the first try, however you will notice that the loop still runs once

        return iteration;
    }
}
```

Exercise

a) Observe and deploy the contract given in the above example.
b) If you input '18' during the function call of the 'iterate()' function, what value will be returned to you?
c) If you input '0' during the function call of the 'iterate()' function, why is '0' value not returned to you?

Exercise

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

contract Loop{


    uint[] public arr;

    function ODDNum(uint max) public{

        uint i = 1;

        while(i <= max){

        if (i % 2 != 0)
            arr.push(i);
            i++;
        }
    }

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

a) Deploy and observe the contract given above.
b) Convert the **while** loop to a **do-while** loop and then implement the modified contract.

3 For Loop

- The for loop is a compact and commonly used loop when the number of iterations is known in advance.
- It consists of three parts: **initialization**, **condition**, and **iteration** statement.
- The loop continues executing as long as the condition is true.

Syntax:

```
for (initialization; condition; iteration statement) {
    // Code to be executed
}
```

Example

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

contract ForLoop{

    uint[] public numtable;
    function table(uint num) public{

        uint temp = num;
        for(uint i = 1; i <= 10; i++){

            numtable.push(num);
            num += temp;
        }
    }

    function get() public view returns(uint[] memory){

        return numtable;
    }

    function resetarr() public{

        delete numtable;
    }
}
```

Exercise

a) Run and observe the contract given in the above example.

b) Run and observe all the functions, through your observation explain what does the ‘resetarr()’ function do.

c) If you want to get the table of a number till ‘20’, instead of ‘10’, what change will you make in the ‘table()’ function.

d) In the **for** loop, if you swap **i++** in the iteration part and **num += temp** at the final line of the loop body, what will happen?

Exercise

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

contract ForLoop{

    int[5] public arr = [-9, 8, 7, -2, 1];
    int[] public positarr;

    function positivepart() public{

        for(uint i = 0; i < arr.length; i++){

            if(arr[i] >= 0){
                positarr.push(arr[i]);
            }
        }
    }
    function getposit() public view returns(int[] memory){

        return positarr;
    }
}
```

a) Run and observe the above contract. Explain what it does.

b) Create an **int** type dynamic array named **negarr** in the same contract.

c) Create another **function** named 'negativepart()' in the same contract that separates the negative part of **arr** and pushes it into the **negarr** array.

d) Create a getter function for your **negarr** array.

Exercise

A function named 'createFact()' is created in the contract given below which takes a **uint** value 'n' as input. The function calculates and returns the factorial of 'n'. Insert a **for** loop in the 'createFact()' function that calculates the Factorial of 'n' and stores it in 'factvalue' variable.

Factorial: A factorial is a number that is the product of a number and all the numbers below it.

e.g: Factorial of '4' will be: 4*3*2*1 = 24

It should also be noted that that factorial of '0' and '1' is 1.

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

contract Factorial{

    function createFact(uint n) public pure returns(uint){


        uint factvalue;

        if(n == 0){

            factvalue = 1;
            return factvalue;
        }

        //******** Insert For Loop Here***********



        return factvalue;
    }
}
```

4 Nested Loops

- A nested while loop refers to any type of loop that is defined inside any other loop.
- Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at **n** times.
- You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a **'for'** loop.

Syntax of Nested loop

```
Outer_loop

{

  Inner_loop

    {

    // inner loop statements.

    }

  // outer loop statements.

}
```

EXERCISE:

Recall the concept of Merkle tree from module 1, instead of hashing just use summation. The root node should reflect the total sum of leaf nodes at level 0.

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


contract MerkleTree {
//    Sum(H1,H2,H3,H4)
//        /      \
//  Sum(H1,H2)  Sum(H3,H4)
//    /  \        /  \
//  H1    H2     H3    H4

  function func_name() {
      While(...){
            For(...){
                  // Body
            }
      }
}

}
```