SMART CONTRACT

A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible.

The concept of smart contracts was first proposed by Nick Szabo in 1994. Szabo is a legal scholar and cryptographer known for laying the groundwork for digital currency.

```
pragma solidity >=0.4.0 <0.6.0;
contract SimpleStorage {
  uint storedData;
  function set(uint x) public {
    storedData = x;
  }
  function get() public view returns (uint) {
    return storedData;
  }
}
```

Pragma

The first line is a pragma directive which tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality up to, but not including, version 0.6.0.

A pragma directive is always local to a source file and if you import another file, the pragma from that file will not automatically apply to the importing file.

So a pragma for a file which will not compile earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 will be written as follows −

pragma solidity ^0.4.0;

Here the second condition is added by using ^.

Contract

A Solidity contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereumblockchain.

The line uintstoredData declares a state variable called storedData of type uint and the functions set and get can be used to modify or retrieve the value of the variable.

Importing Files

Though above example does not have an import statement but Solidity supports import statements that are very similar to those available in JavaScript.

The following statement imports all global symbols from "filename".

import "filename";

The following example creates a new global symbol symbolName whose members are all the global symbols from "filename".

import * as symbolName from "filename";

To import a file x from the same directory as the current file, use import "./x" as x;. If you use import "x" as x; instead, a different file could be referenced in a global "include directory".

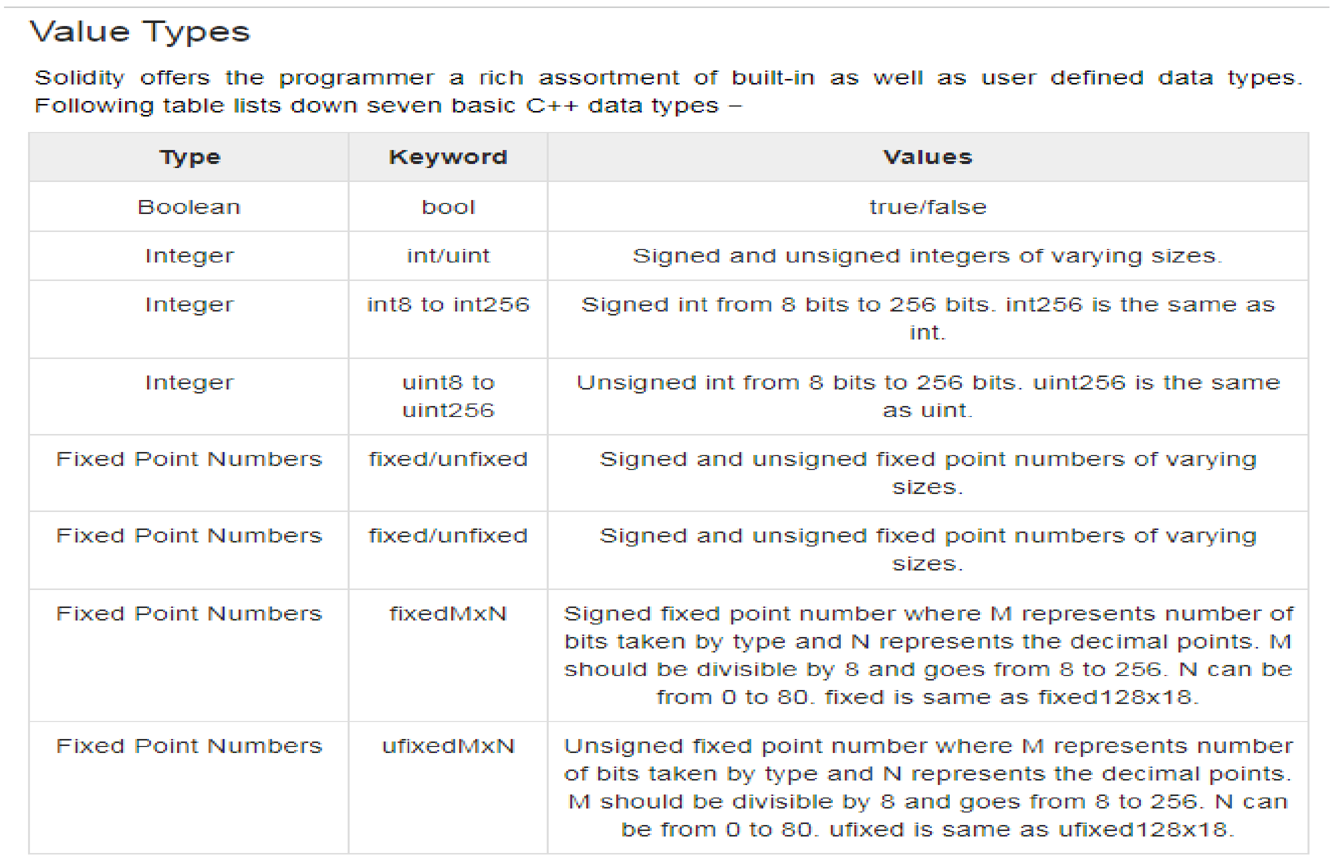
Comments

Solidity supports both C-style and C++-style comments, Thus −

Any text between a // and the end of a line is treated as a comment and is ignored by Solidity Compiler.

Any text between the characters /* and */ is treated as a comment. This may span multiple lines.

Variables



Value Types

Solidity offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

Type	Keyword	Values
Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes.
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is the same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is the same as uint.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixedMxN	Signed fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. fixed is same as fixed128x18.
Fixed Point Numbers	ufixedMxN	Unsigned fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. ufixed is same as ufixed128x18.

Solidity supports three types of variables.

State Variables − Variables whose values are permanently stored in a contract storage.

Local Variables − Variables whose values are present till function is executing.

Global Variables − Special variables exists in the global namespace used to get information about the blockchain.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration. Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

State Variable

Variables whose values are permanently stored in a contract storage.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData;      // State variable
   constructor() public {
      storedData = 10;   // Using State variable
   }
```

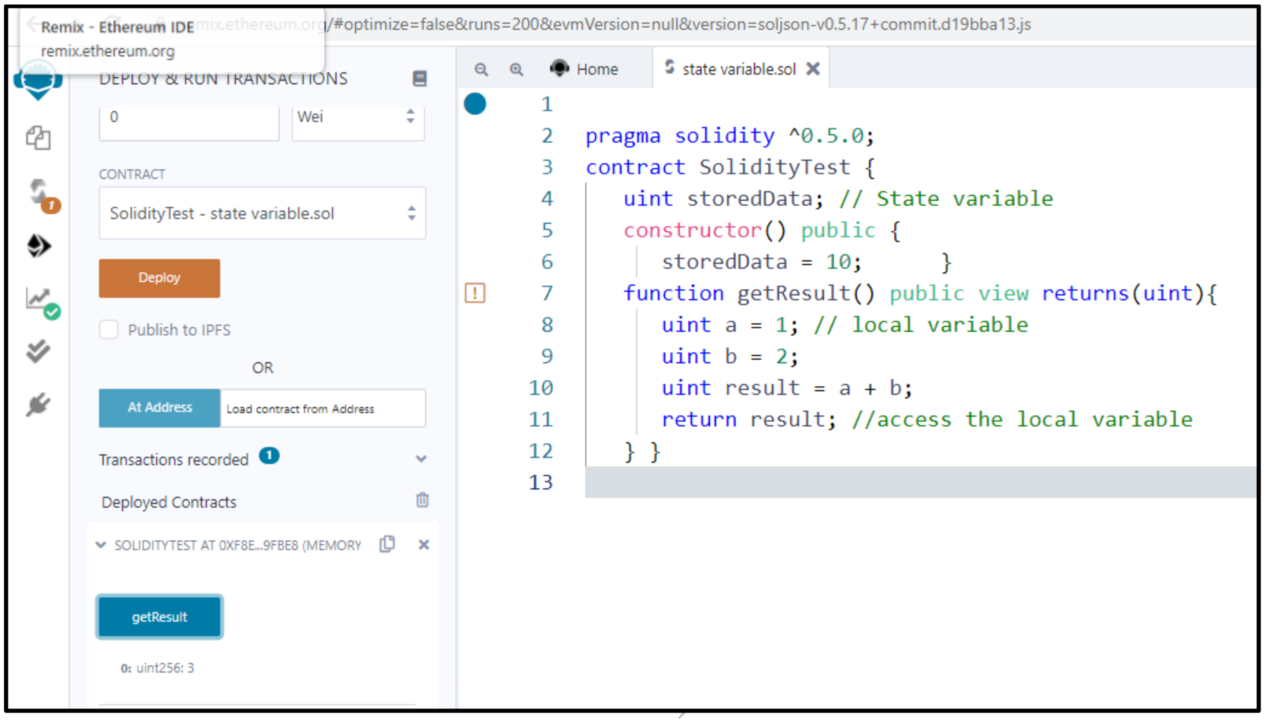
Local Variable

Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

Program

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;    }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   }
  }
```

Output


Remix - Ethereum IDE
remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
Home
state variable.sol
0
Wei
CONTRACT
SolidityTest - state variable.sol
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 1
Deployed Contracts
SOLIDITYTEST AT 0XF8E...9FBE8 (MEMORY
getResult

```
0: uint256: 3
```

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;      }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   } }
```

FUNCTION

A function is basically a group of code that can be reused anywhere in the program, which generally saves the excessive use of memory and decreases the runtime of the program. Creating a function reduces the need of writing the same code over and over again. With the help of functions, a program can be divided into many small pieces of code for better understanding and managing.

Declaring a Function

In Solidity a function is generally defined by using the function keyword, followed by the name of the function which is unique and does not match with any of the reserved keywords. A function can also have a list of parameters containing the name and data type of the parameter. The return value of a function is optional but in solidity, the return type of the function is defined at the time of declaration.

```
function function_name(parameter_list) scope returns(return_type) {

      // block of code

}
```

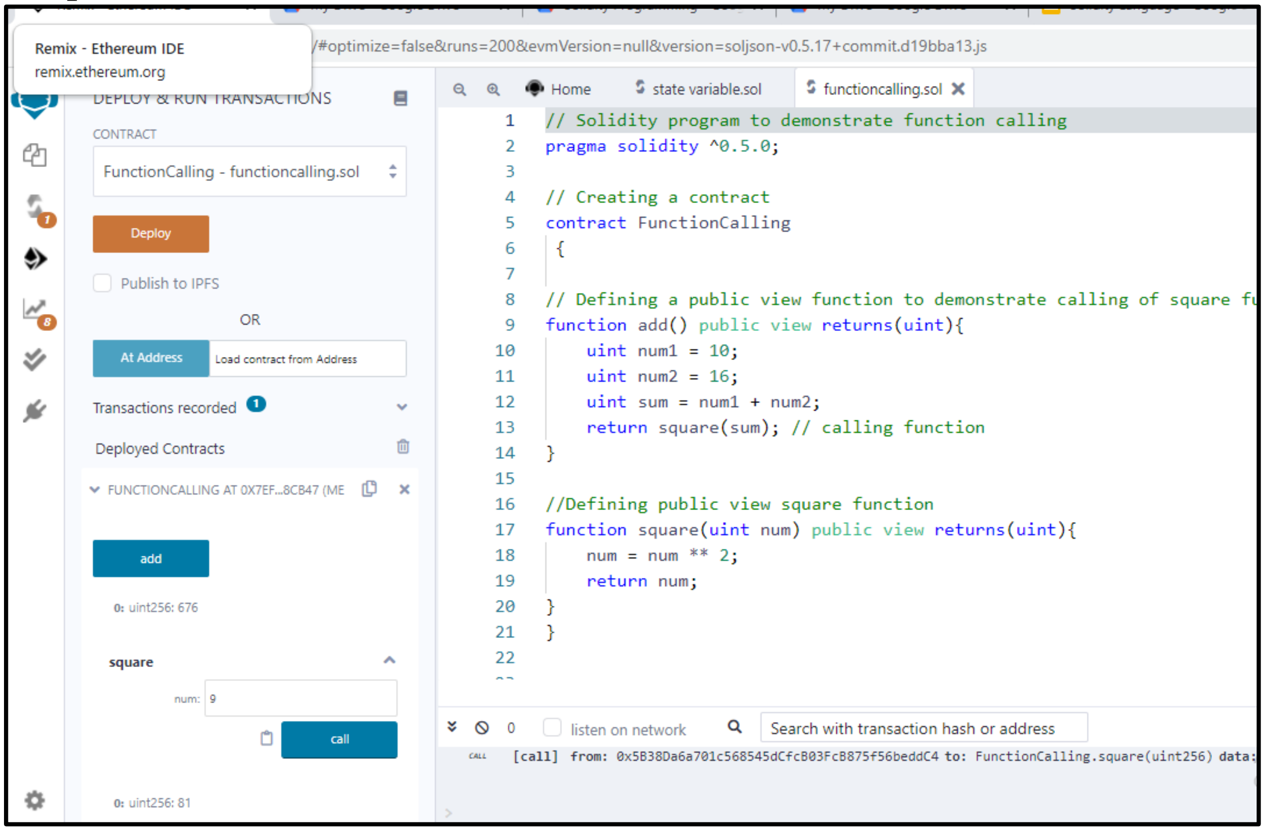
Function Calling

A function is called when the user wants to execute that function. In Solidity the function is simply invoked by writing the name of the function where it has to be called. Different parameters can be passed to function while calling, multiple parameters can be passed to a function by separating with a comma.

Program

```
// Solidity program to demonstrate function calling

pragma solidity ^0.5.0;


// Creating a contract

contract FunctionCalling {


// Defining a public view function to demonstrate calling of sqrt function
function add() public view returns(uint){
        uint num1 = 10;
        uint num2 = 16;
        uint sum = num1 + num2;
        return sqrt(sum); // calling function
}

//Defining public view sqrt function
function sqrt(uint num) public view returns(uint){
        num = num ** 2;
        return num;
}
}
```

Output


Remix - Ethereum IDE
remix.ethereum.org
/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
Home
state variable.sol
functioncalling.sol

CONTRACT
FunctionCalling - functioncalling.sol
Deploy
Publish to IPFS
OR
At Address | Load contract from Address
Transactions recorded 1
Deployed Contracts
FUNCTIONCALLING AT 0X7EF...8CB47 (ME
add
0: uint256: 676
square
num: 9
call
0: uint256: 81

```
// Solidity program to demonstrate function calling
pragma solidity ^0.5.0;

// Creating a contract
contract FunctionCalling
 {

// Defining a public view function to demonstrate calling of square fu
function add() public view returns(uint){
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    return square(sum); // calling function
}

//Defining public view square function
function square(uint num) public view returns(uint){
    num = num ** 2;
    return num;
}
}
```

```
0    listen on network    Search with transaction hash or address
CALL    [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: FunctionCalling.square(uint256) data:
```

Return Statements

A return statement is an optional statement provided by solidity. It is the last statement of the function, used to return the values from the functions. In Solidity, more than one value can be returned from a function. To return values from the function, the data types of the return values should be defined at function declaration.

Program

// Solidity program to demonstrate return statements

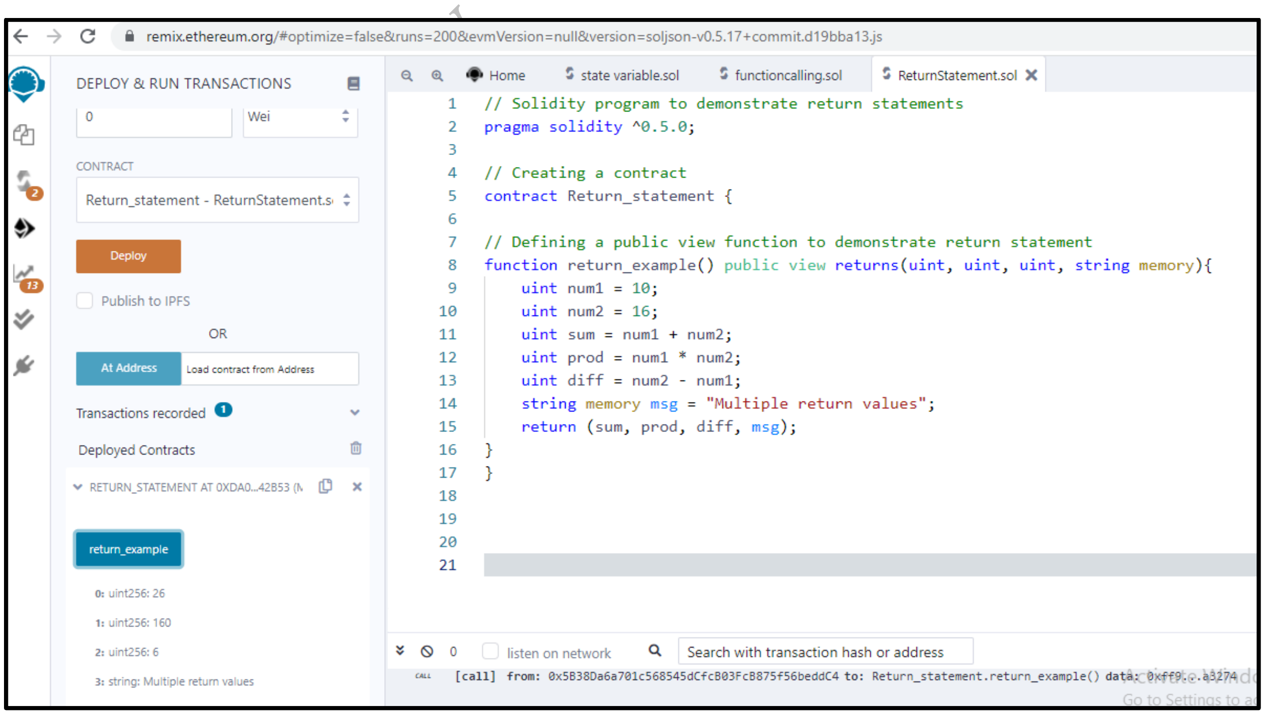
pragma solidity ^0.5.0;

// Creating a contract

contract Return_statement {

```
// Defining a public view function to demonstrate return statement
function return_example() public view returns(uint, uint, uint, string memory){
        uint num1 = 10;
        uint num2 = 16;
        uint sum = num1 + num2;
        uint prod = num1 * num2;
        uint diff = num2 - num1;
        string memory msg = "Multiple return values";
        return (sum, prod, diff, msg);
}
}
```

Output


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
0
Wei
CONTRACT
Return_statement - ReturnStatement.s
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 1
Deployed Contracts
RETURN_STATEMENT AT 0XDA0...42B53 (M
return_example
Home
state variable.sol
functioncalling.sol
ReturnStatement.sol
listen on network
Search with transaction hash or address

```
0: uint256: 26
1: uint256: 160
2: uint256: 6
3: string: Multiple return values
```

```
// Solidity program to demonstrate return statements
pragma solidity ^0.5.0;

// Creating a contract
contract Return_statement {

// Defining a public view function to demonstrate return statement
function return_example() public view returns(uint, uint, uint, string memory){
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    uint prod = num1 * num2;
    uint diff = num2 - num1;
    string memory msg = "Multiple return values";
    return (sum, prod, diff, msg);
}
}
```

```
CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Return_statement.return_example() data: 0xff9...a3274
```

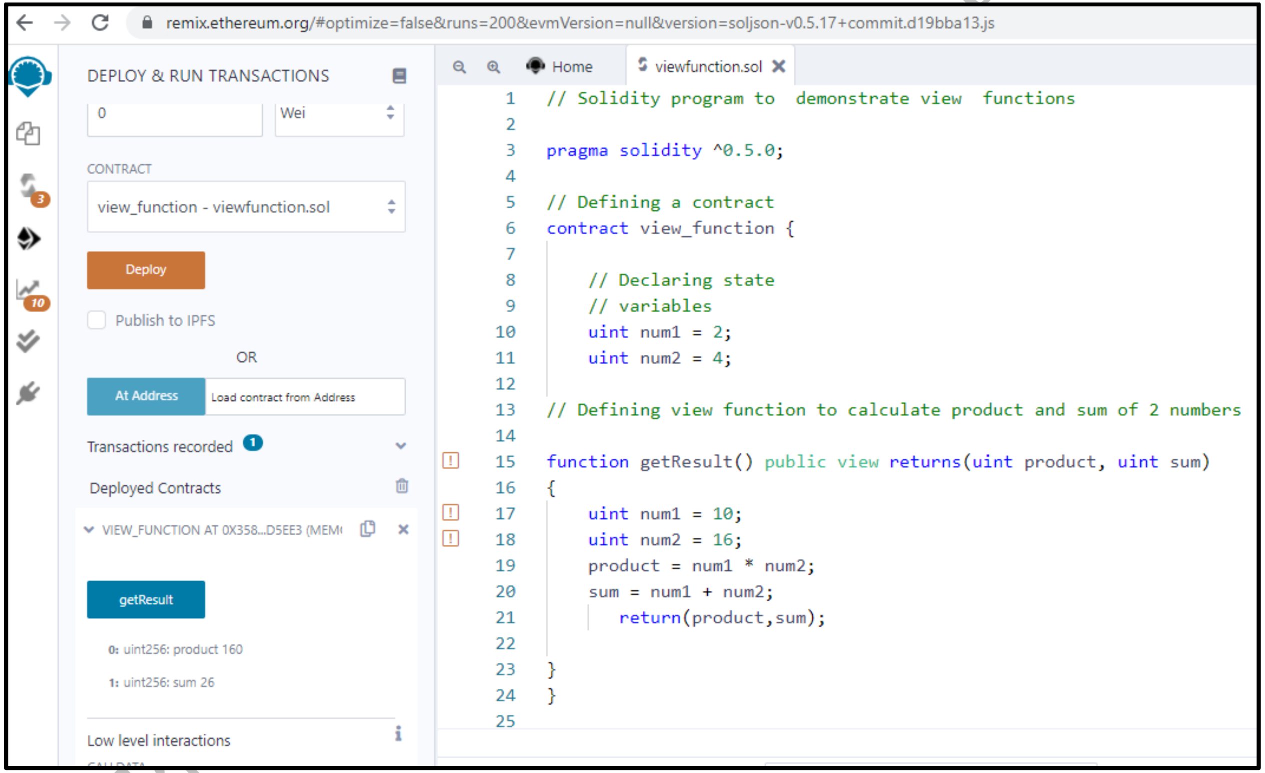
Solidity – View and Pure Functions (Types of Function)

The view functions are read-only function, which ensures that state variables cannot be modified after calling them. If the statements which modify state variables, emitting events, creating other contracts, using selfdestruct method, transferring ethers via calls, Calling a function which is not 'view or pure', using low-level calls, etc are present in view functions then the compiler throw a warning in such cases. By default, a get method is view function.

Program

```
// Solidity program to  demonstrate view  functions

pragma solidity ^0.5.0;

// Defining a contract
contract view_function {

        // Declaring state
        // variables
        uint num1 = 2;
        uint num2 = 4;

// Defining view function to calculate product and sum of 2 numbers

function getResult() public view returns(uint product, uint sum)
{
        uint num1 = 10;
        uint num2 = 16;
        product = num1 * num2;
        sum = num1 + num2;
      return(product,sum);
   }
   }
```

Output


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
0
Wei
CONTRACT
view_function - viewfunction.sol
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 1
Deployed Contracts
VIEW_FUNCTION AT 0X358...D5EE3 (MEMORY)
getResult
Low level interactions
Home
viewfunction.sol

```
0: uint256: product 160
1: uint256: sum 26
```

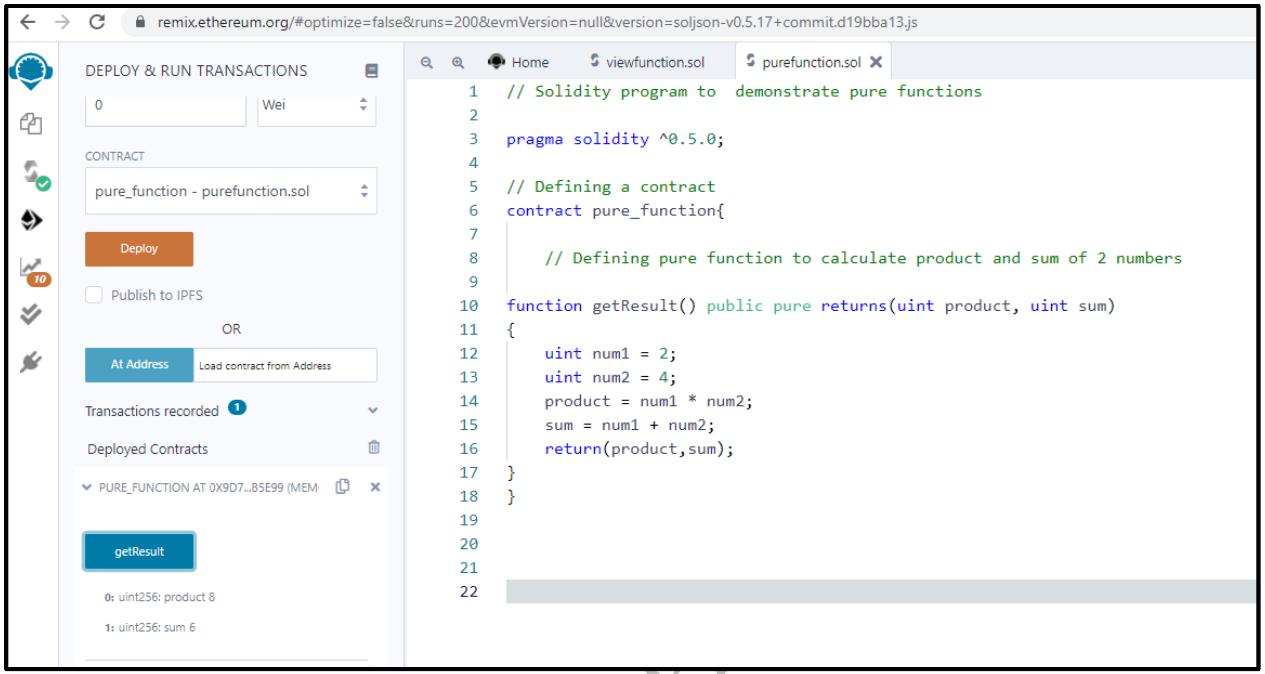
```
// Solidity program to  demonstrate view  functions

pragma solidity ^0.5.0;

// Defining a contract
contract view_function {

    // Declaring state
    // variables
    uint num1 = 2;
    uint num2 = 4;

// Defining view function to calculate product and sum of 2 numbers

function getResult() public view returns(uint product, uint sum)
{
    uint num1 = 10;
    uint num2 = 16;
    product = num1 * num2;
    sum = num1 + num2;
      return(product,sum);

}
}
```

The **pure functions** do not read or modify the state variables, which returns the values only using the parameters passed to the function or local variables present in it. If the statements which read the state variables, access the address or balance, accessing any global variable block or msg, calling a function which is not pure, etc are present in pure functions then the compiler throws a warning in such cases.

Program:

```
// Solidity program to  demonstrate pure functions

pragma solidity ^0.5.0;

// Defining a contract
contract pure_function{

        // Defining pure function to calculate product and sum of 2 numbers

function getResult() public pure returns(uint product, uint sum)
{
        uint num1 = 2;
        uint num2 = 4;
        product = num1 * num2;
        sum = num1 + num2;
        return(product,sum);
}
}
```

Output:



remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

DEPLOY & RUN TRANSACTIONS

CONTRACT: pure_function - purefunction.sol

PURE_FUNCTION AT 0X9D7...B5E99 (MEM

getResult

```
0: uint256: product 8
1: uint256: sum 6
```

```
// Solidity program to  demonstrate pure functions

pragma solidity ^0.5.0;

// Defining a contract
contract pure_function{

    // Defining pure function to calculate product and sum of 2 numbers

function getResult() public pure returns(uint product, uint sum)
{
    uint num1 = 2;
    uint num2 = 4;
    product = num1 * num2;
    sum = num1 + num2;
    return(product,sum);
}
}
```

LOOP

Loops are used when we have to perform an action over and over again. While writing a contract there may be a situation when we have to do some action repeatedly, In this situation, loops are implemented to reduce the number of lines of the statements. Solidity supports following loops too ease down the programming pressure.

While Loop

This is the most basic loop in solidity, Its purpose is to execute a statement or block of statements repeatedly as far as the condition is true and once the condition becomes false the loop terminates.

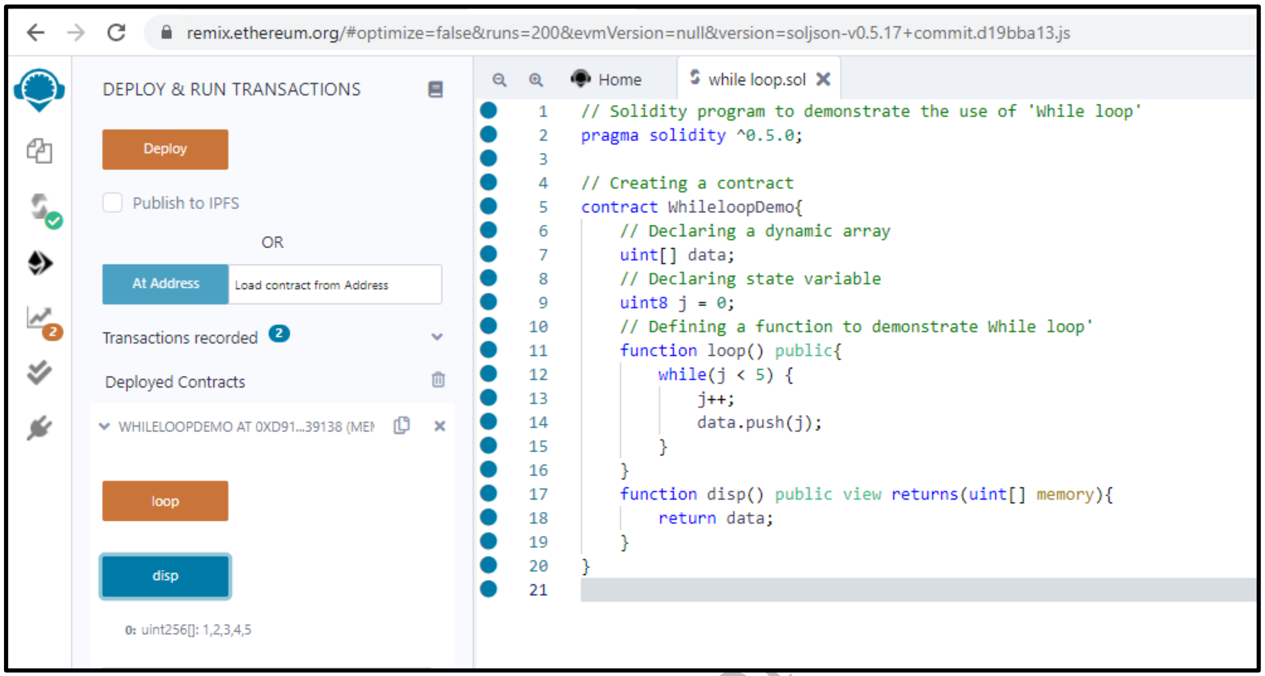
Syntax:

```
while (condition) {
   statement or block of code to be executed if the condition is True
}
```

Program:

```
// Solidity program to demonstrate the use of 'While loop'
pragma solidity ^0.5.0;

// Creating a contract
contract WhileloopDemo{
        // Declaring a dynamic array
        uint[] data;
        // Declaring state variable
        uint8 j = 0;
        // Defining a function to demonstrate While loop'
        function loop() public{
           while(j < 5) {
                   j++;
                   data.push(j);
           }
        }
        function disp() public view returns(uint[] memory){
        return data;
        }
}
```

Output:


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 2
Deployed Contracts
WHILELOOPDEMO AT 0XD91...39138 (MEI
loop
disp
Home
while loop.sol

```
// Solidity program to demonstrate the use of 'While loop'
pragma solidity ^0.5.0;

// Creating a contract
contract WhileloopDemo{
    // Declaring a dynamic array
    uint[] data;
    // Declaring state variable
    uint8 j = 0;
    // Defining a function to demonstrate While loop'
    function loop() public{
        while(j < 5) {
            j++;
            data.push(j);
        }
    }
    function disp() public view returns(uint[] memory){
        return data;
    }
}
```

```
0: uint256[]: 1,2,3,4,5
```

Do-While Loop

This loop is very similar to while loop except that there is a condition check which happens at the end of loop i.e. the loop will always execute at least one time even if the condition is false.

Syntax:

do

{

  block of statements to be executed

} while (condition);

Program:

// Solidity program to demonstrate the use of 'Do-While loop'

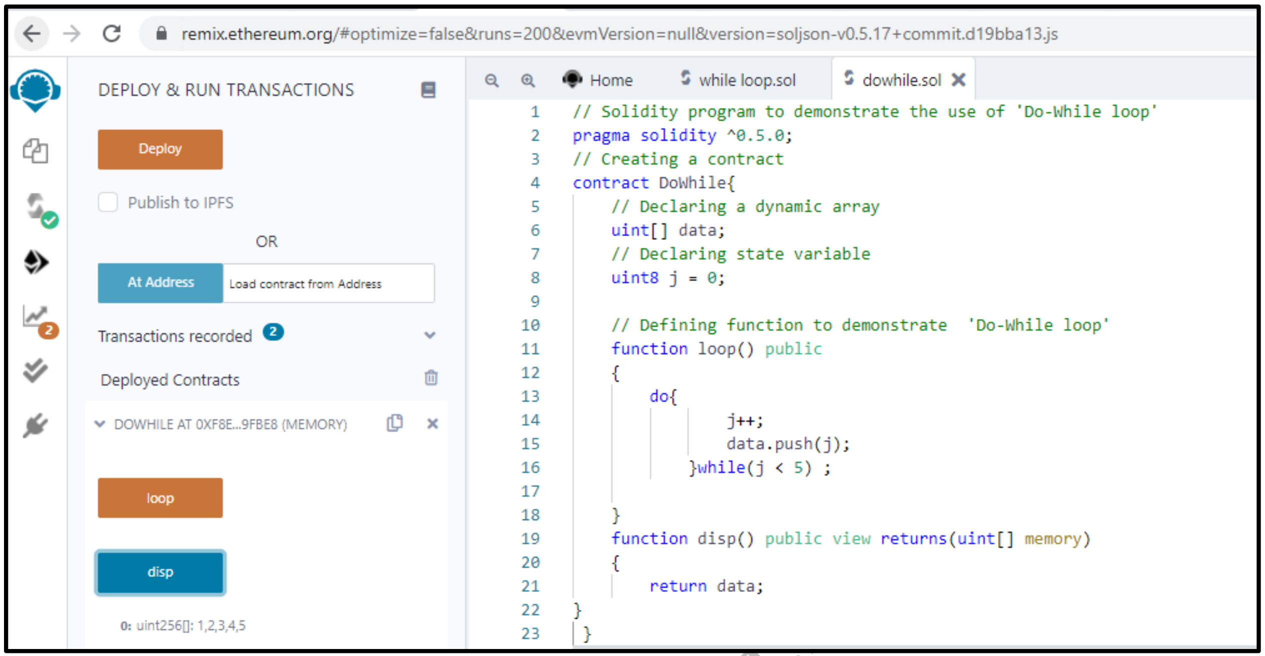
pragma solidity ^0.5.0;

// Creating a contract

contract DoWhile{

```
        // Declaring a dynamic array
        uint[] data;
        // Declaring state variable
        uint8 j = 0;

        // Defining function to demonstrate  'Do-While loop'
        function loop() public
    {
          do{
              j++;
                    data.push(j);
             }while(j < 5) ;

        }
    function disp() public view returns(uint[] memory)
    {
        return data;
}
 }
```

Output:



remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

```
// Solidity program to demonstrate the use of 'Do-While loop'
pragma solidity ^0.5.0;
// Creating a contract
contract DoWhile{
    // Declaring a dynamic array
    uint[] data;
    // Declaring state variable
    uint8 j = 0;

    // Defining function to demonstrate  'Do-While loop'
    function loop() public
    {
        do{
                j++;
                data.push(j);
            }while(j < 5) ;

    }
    function disp() public view returns(uint[] memory)
    {
        return data;
    }
}
```

```
0: uint256[]: 1,2,3,4,5
```

For Loop

This is the most compact way of looping. It takes three arguments separated by a semi-colon to run. The first one is 'loop initialization' where the iterator is initialized with starting value, this statement is executed before the loop starts. Second is 'test statement' which checks whether the condition is true or not, if the condition is true the loop executes else terminates. The third one is the 'iteration statement' where the iterator is increased or decreased. Below is the syntax of for loop :

Syntax:

```
for (initialization; test condition; iteration statement) {
      statement or block of code to be executed if the condition is True
}
```

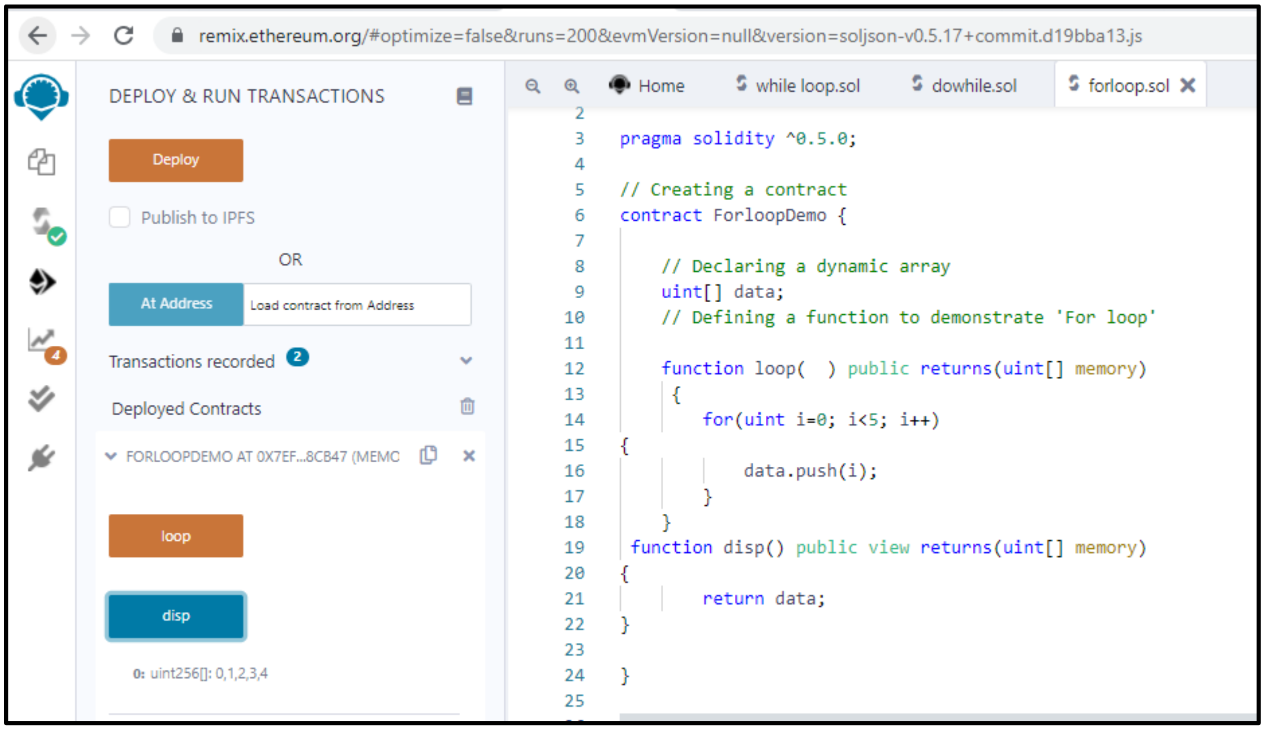
Program:

// Solidity program to demonstrate the use of 'For loop'

pragma solidity ^0.5.0;

```
// Creating a contract
contract ForloopDemo {

         // Declaring a dynamic array
         uint[] data;
         // Defining a function to demonstrate 'For loop'

         function loop(        ) public returns(uint[] memory)
     {
                   for(uint i=0; i<5; i++)
                   {
                             data.push(i);                    }     }
 function disp() public view returns(uint[] memory)
{
         return data;
}
}
```

Output:


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 2
Deployed Contracts
FORLOOPDEMO AT 0X7EF...8CB47 (MEMO
loop
disp
Home
while loop.sol
dowhile.sol
forloop.sol

0: uint256[]: 0,1,2,3,4

```

pragma solidity ^0.5.0;

// Creating a contract
contract ForloopDemo {

    // Declaring a dynamic array
    uint[] data;
    // Defining a function to demonstrate 'For loop'

    function loop(  ) public returns(uint[] memory)
     {
        for(uint i=0; i<5; i++)
{
            data.push(i);
        }
    }
 function disp() public view returns(uint[] memory)
{
        return data;
}

}

```

REFERENCE TYPES

Reference type variables store the location of the data. They don't share the data directly. With the help of reference type, two different variables can refer to the same location where any change in one variable can affect the other one. Reference type in solidity are listed below:

Arrays: An array is a group of variables of the same data type in which variable has a particular location known as an index. By using the index location, the desired variable can be accessed. The array size can be fix or dynamic.

Struct: Solidity allows users to create and define their own type in the form of structures. The structure is a group of different types even though it's not possible to contain a member of its own type. The structure is a reference type variable which can contain both value type and reference type

Mapping: Mapping is a most used reference type, that stores the data in a key-value pair where a key can be any value types. It is like a hashtable or dictionary as in any other programming language, where data can be retrieved by key.

SOLIDITY – STRUCT

Solidity allows user to create their own data type in the form of structure. The struct contains a group of elements with a different data type. Generally, it is used to represent a record. To define a structure struct keyword is used, which creates a new data type.

Syntax:

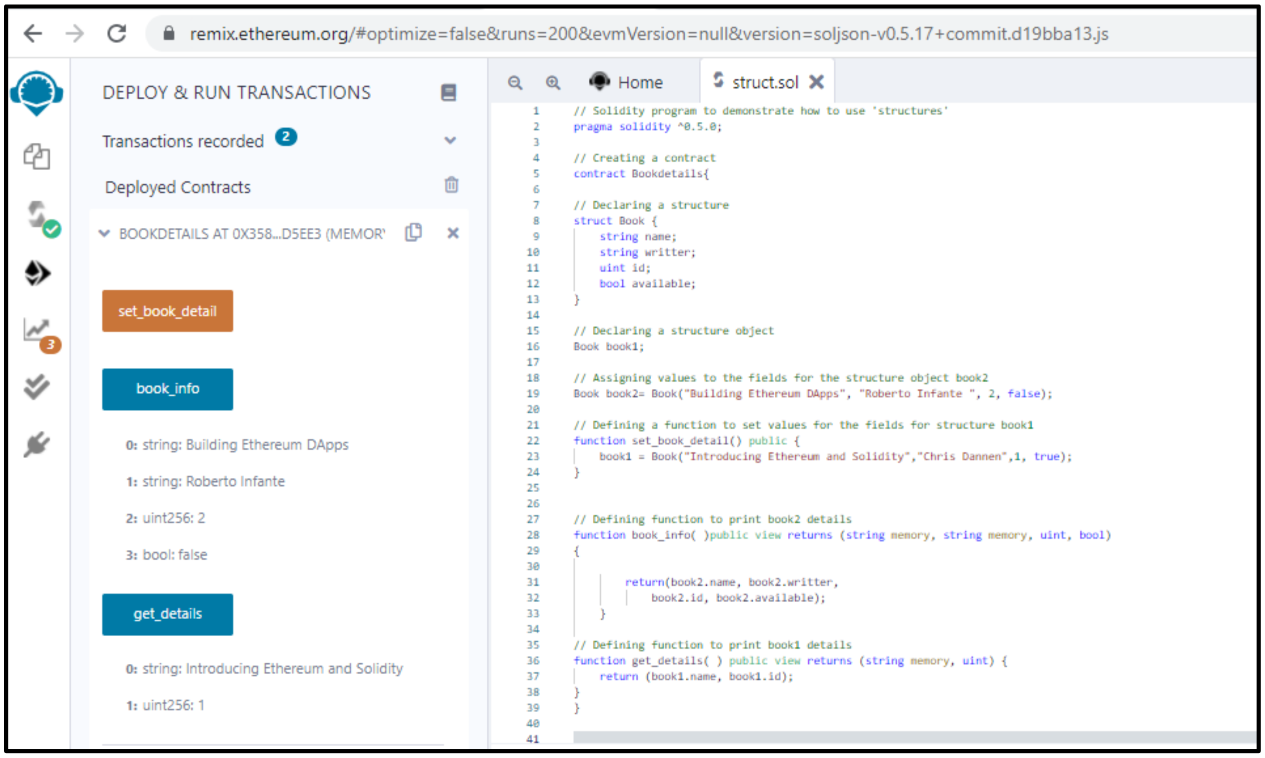
```
struct <structure_name> {
  <data type> variable_1;
  <data type> variable_2;
}
```

For accessing any element of the structure dot operator is used, which separates the struct variable and the element we wish to access. To define the variable of structure data type structure name is used.

Program:

```
// Solidity program to demonstrate how to use 'structures'
pragma solidity ^0.5.0;

// Creating a contract
contract Bookdetails{

// Declaring a structure
struct Book {
        string name;
        string writter;
        uint id;
        bool available;
}

// Declaring a structure object
Book book1;

// Assigning values to the fields for the structure object book2
Book book2= Book("Building Ethereum DApps",      "Roberto Infante ",      2, false);

// Defining a function to set values for the fields for structure book1
function set_book_detail() public {
        book1 = Book("Introducing Ethereum and Solidity","Chris Dannen",1, true);
}


// Defining function to print book2 details
function book_info( )public view returns (string memory, string memory, uint, bool)
{

                return(book2.name, book2.writter,
                        book2.id, book2.available);
        }

// Defining function to print book1 details
function get_details( ) public view returns (string memory, uint) {
        return (book1.name, book1.id);
}
}
```

Output:



remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

```
0: string: Building Ethereum DApps
1: string: Roberto Infante
2: uint256: 2
3: bool: false
```

```
0: string: Introducing Ethereum and Solidity
1: uint256: 1
```

```
// Solidity program to demonstrate how to use 'structures'
pragma solidity ^0.5.0;

// Creating a contract
contract Bookdetails{

// Declaring a structure
struct Book {
    string name;
    string writter;
    uint id;
    bool available;
}

// Declaring a structure object
Book book1;

// Assigning values to the fields for the structure object book2
Book book2= Book("Building Ethereum DApps", "Roberto Infante ", 2, false);

// Defining a function to set values for the fields for structure book1
function set_book_detail() public {
    book1 = Book("Introducing Ethereum and Solidity","Chris Dannen",1, true);
}


// Defining function to print book2 details
function book_info( )public view returns (string memory, string memory, uint, bool)
{

        return(book2.name, book2.writter,
            book2.id, book2.available);
    }

// Defining function to print book1 details
function get_details( ) public view returns (string memory, uint) {
    return (book1.name, book1.id);
}
}
```

SOLIDITY – MAPPINGS

Mapping in Solidity acts like a hash table or dictionary in any other language. These are used to store the data in the form of key-value pairs, a key can be any of the built-in data types but reference types are not allowed while the value can be of any type. Mappings are mostly used to associate the unique Ethereum address with the associated value type.

Syntax:

mapping(key => value) <access specifier> <name>;

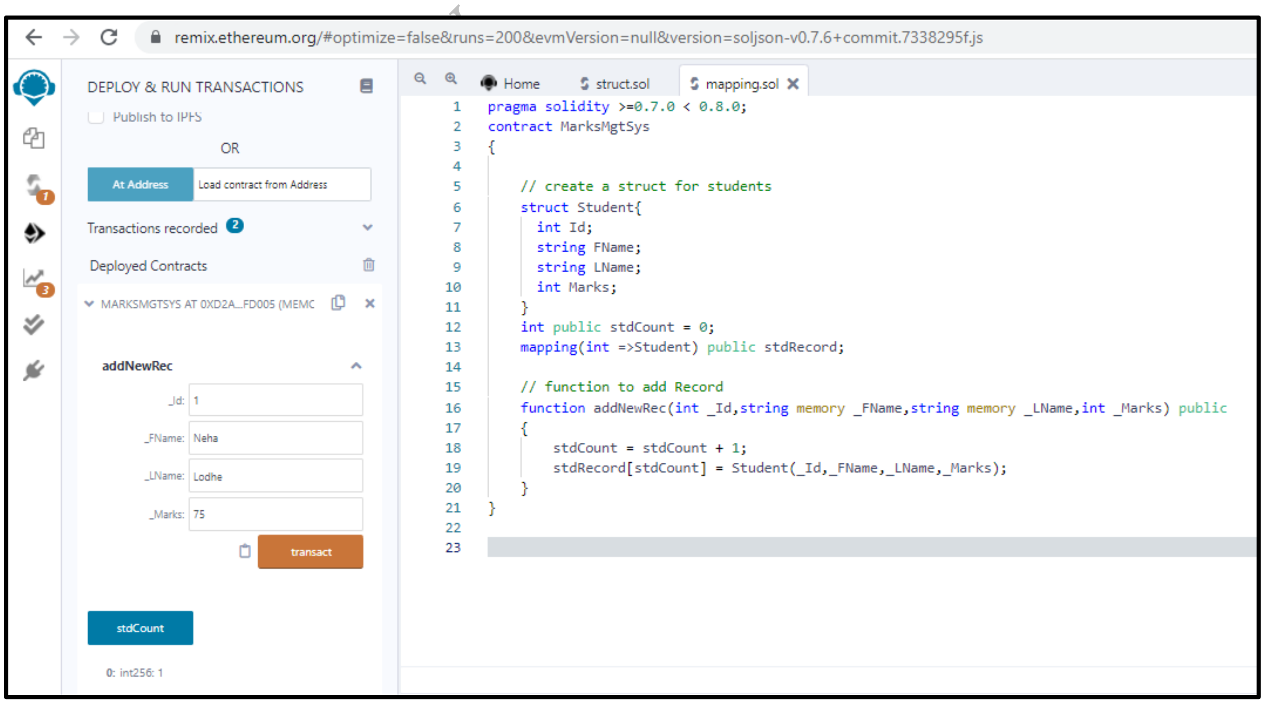
Creating a Mapping

Mapping is defined as any other variable type, which accepts a key type and a value type.

Program:

```
pragma solidity >=0.7.0 < 0.8.0;
contract MarksMgtSys
{

   // create a struct for students
   struct Student{
     int Id;
     string FName;
     string LName;
     int Marks;
   }
   int public stdCount = 0;
   mapping(int =>Student) public stdRecord;

   // function to add Record
   function addNewRec(int _Id,string memory _FName,string memory _LName,int _Marks)
public
   {
       stdCount = stdCount + 1;
       stdRecord[stdCount] = Student(_Id,_FName,_LName,_Marks);
   }
}
```

Output:



```
remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.7.6+commit.7338295f.js
```

DEPLOY & RUN TRANSACTIONS

At Address | Load contract from Address

Transactions recorded 2

Deployed Contracts

MARKSMGTSYS AT 0XD2A...FD005 (MEMC

addNewRec

_Id: 1

_FName: Neha

_LName: Lodhe

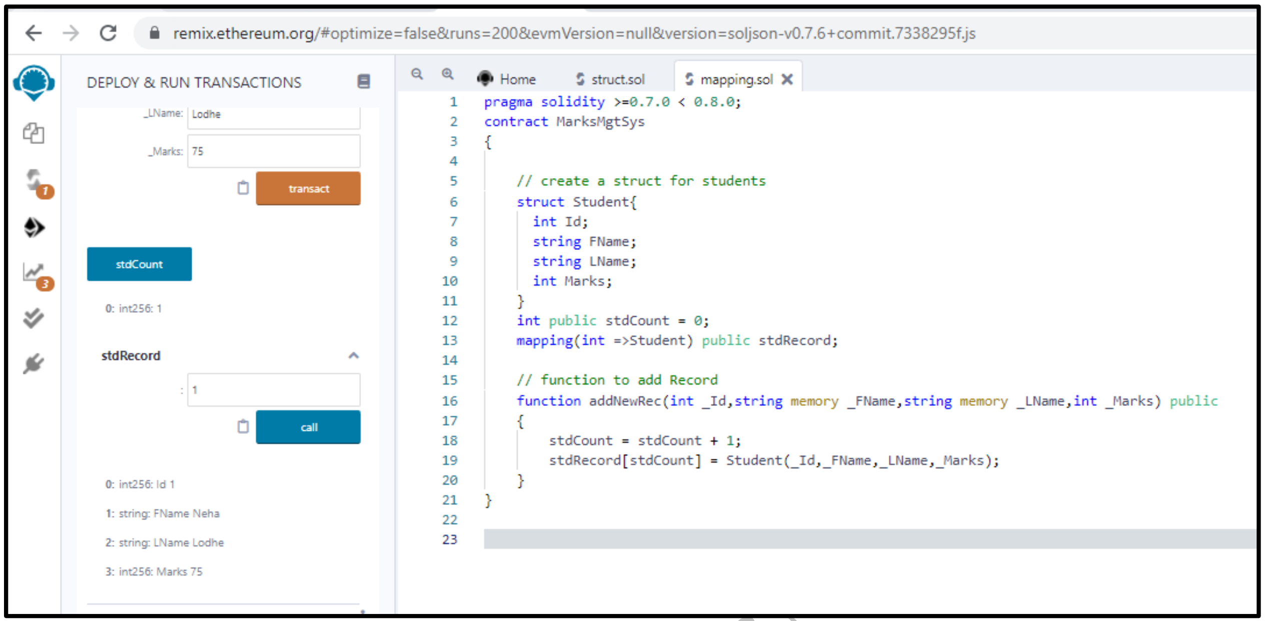
_Marks: 75

transact

stdCount

0: int256: 1

```
pragma solidity >=0.7.0 < 0.8.0;
contract MarksMgtSys
{

    // create a struct for students
    struct Student{
      int Id;
      string FName;
      string LName;
      int Marks;
    }
    int public stdCount = 0;
    mapping(int =>Student) public stdRecord;

    // function to add Record
    function addNewRec(int _Id,string memory _FName,string memory _LName,int _Marks) public
    {
        stdCount = stdCount + 1;
        stdRecord[stdCount] = Student(_Id,_FName,_LName,_Marks);
    }
}
```


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.7.6+commit.7338295f.js
DEPLOY & RUN TRANSACTIONS
Home
struct.sol
mapping.sol
transact
stdCount
stdRecord
call

_LName: Lodhe
_Marks: 75

0: int256: 1

: 1

```
0: int256: Id 1
1: string: FName Neha
2: string: LName Lodhe
3: int256: Marks 75
```

```
pragma solidity >=0.7.0 < 0.8.0;
contract MarksMgtSys
{

    // create a struct for students
    struct Student{
      int Id;
      string FName;
      string LName;
      int Marks;
    }
    int public stdCount = 0;
    mapping(int =>Student) public stdRecord;

    // function to add Record
    function addNewRec(int _Id,string memory _FName,string memory _LName,int _Marks) public
    {
        stdCount = stdCount + 1;
        stdRecord[stdCount] = Student(_Id,_FName,_LName,_Marks);
    }
}
```

SOLIDITY – ARRAYS

Arrays are data structures that store the fixed collection of elements of the same data types in which each and every element has a specific location called index.

Instead of creating numerous individual variables of the same type, we just declare one array of the required size and store the elements in the array and can be accessed using the index.

In Solidity, an array can be of fixed size or dynamic size. Arrays have a continuous memory location, where the lowest index corresponds to the first element while the highest represents the last

Creating an Array

To declare an array in Solidity, the data type of the elements and the number of elements should be specified. The size of the array must be a positive integer and data type should be a valid Solidity type

Syntax:

<data type> <array name>[size] = <initialization>

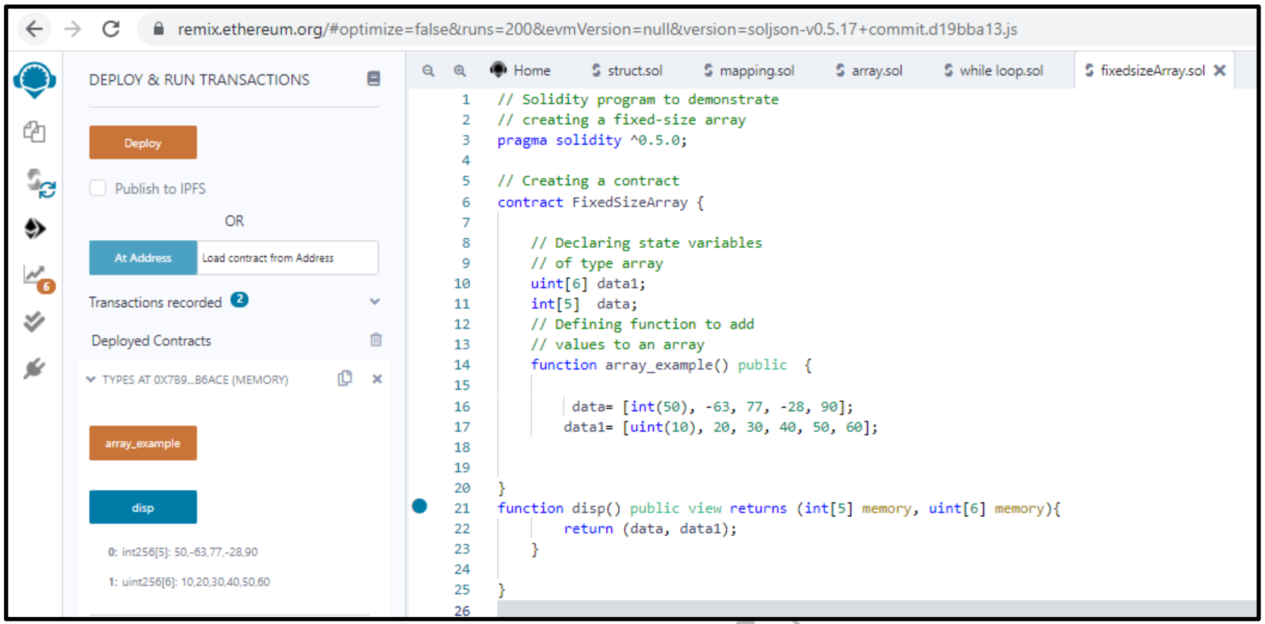
Fixed-size Arrays

The size of the array should be predefined. The total number of elements should not exceed the size of the array. If the size of the array is not specified then the array of enough size is created which is enough to hold the initialization.

Program:

```
// Solidity program to demonstrate
// creating a fixed-size array
pragma solidity ^0.5.0;

// Creating a contract
contract FixedSizeArray {

        // Declaring state variables
        // of type array
        uint[6] data1;
        int[5]  data;
        // Defining function to add
        // values to an array
        function array_example() public  {

                data= [int(50), -63, 77, -28, 90];
                data1= [uint(10), 20, 30, 40, 50, 60];
        }
function disp() public view returns (int[5] memory, uint[6] memory){
        return (data, data1);
        }
}
```

Output:



```
remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
```

```
// Solidity program to demonstrate
// creating a fixed-size array
pragma solidity ^0.5.0;

// Creating a contract
contract FixedSizeArray {

    // Declaring state variables
    // of type array
    uint[6] data1;
    int[5]  data;
    // Defining function to add
    // values to an array
    function array_example() public  {

         data= [int(50), -63, 77, -28, 90];
        data1= [uint(10), 20, 30, 40, 50, 60];


}
function disp() public view returns (int[5] memory, uint[6] memory){
        return (data, data1);
    }

}
```

```
0: int256[5]: 50,-63,77,-28,90
1: uint256[6]: 10,20,30,40,50,60
```

Dynamic Array:

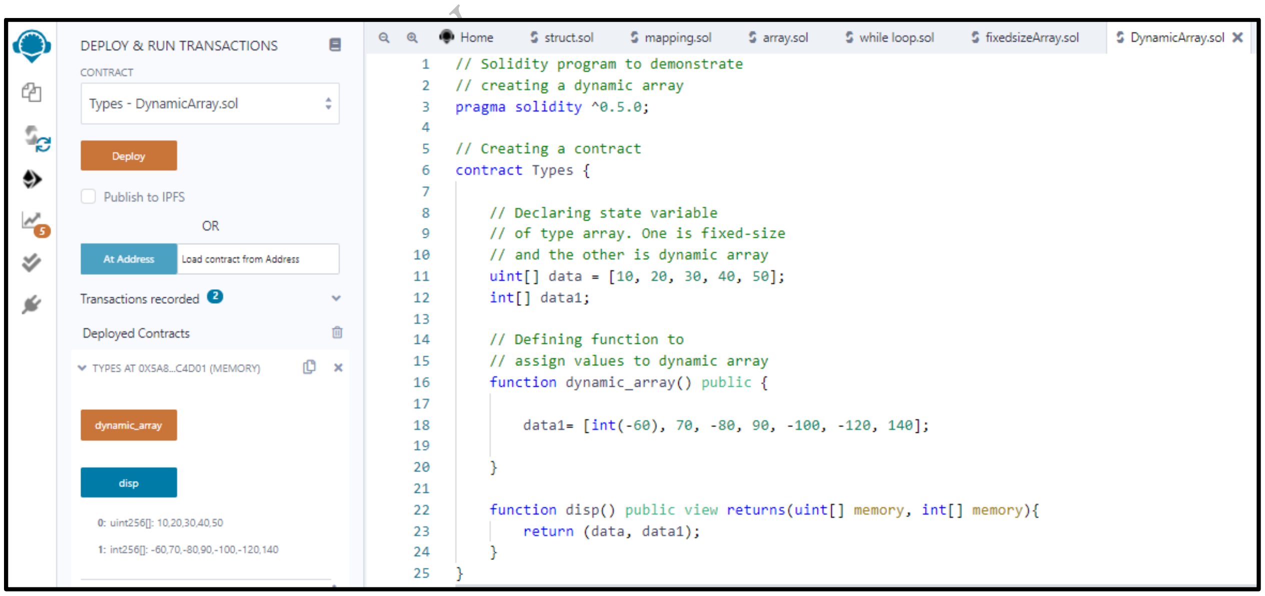
The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined.

Program:

```
// Solidity program to demonstrate
// creating a dynamic array
pragma solidity ^0.5.0;

// Creating a contract
contract Types {

        // Declaring state variable
        // of type array. One is fixed-size
        // and the other is dynamic array
        uint[] data    = [10, 20, 30, 40, 50];
```

```
            int[] data1;

            // Defining function to
            // assign values to dynamic array
            function dynamic_array() public {

                data1= [int(-60), 70, -80, 90, -100, -120, 140];

            }

         function disp() public view returns(uint[] memory, int[] memory){
            return (data, data1);
            }
    }
```

Output:



```
0: uint256[]: 10,20,30,40,50
1: int256[]: -60,70,-80,90,-100,-120,140
```

```
// Solidity program to demonstrate
// creating a dynamic array
pragma solidity ^0.5.0;

// Creating a contract
contract Types {

    // Declaring state variable
    // of type array. One is fixed-size
    // and the other is dynamic array
    uint[] data = [10, 20, 30, 40, 50];
    int[] data1;

    // Defining function to
    // assign values to dynamic array
    function dynamic_array() public {

        data1= [int(-60), 70, -80, 90, -100, -120, 140];

    }

    function disp() public view returns(uint[] memory, int[] memory){
        return (data, data1);
    }
}
```

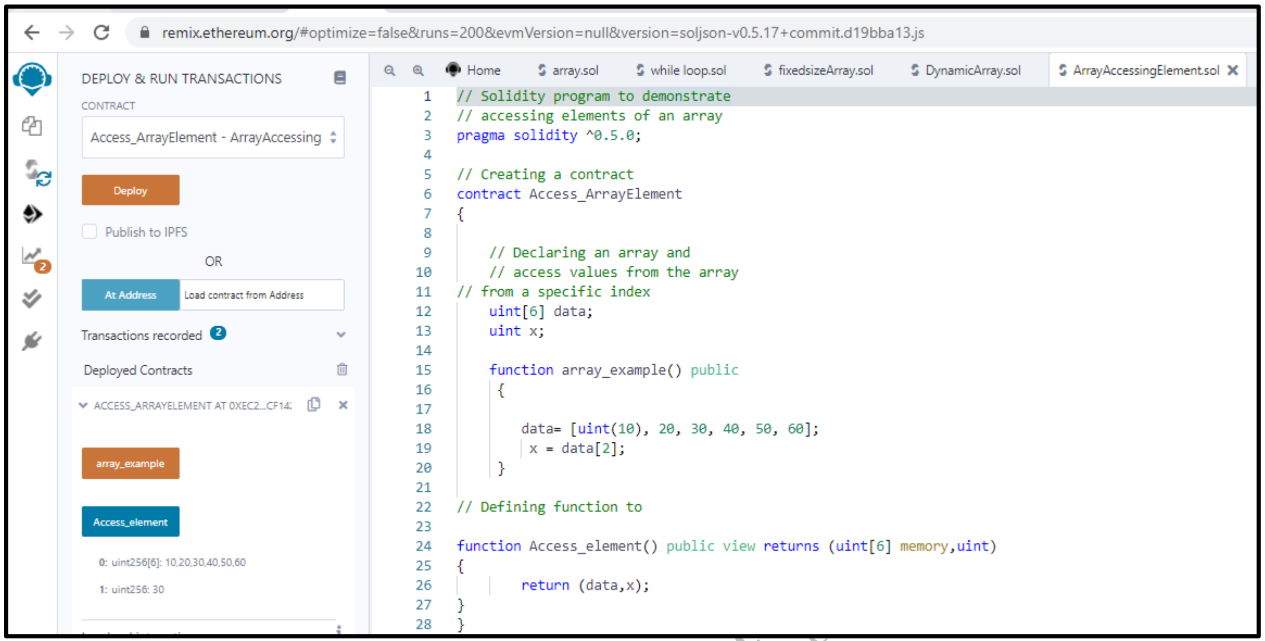
Array Operations

Accessing Array Elements: The elements of the array are accessed by using the index. If you want to access ith element then you have to access (i-1)th index.

Program:

```
// Solidity program to demonstrate
// accessing elements of an array
pragma solidity ^0.5.0;
// Creating a contract
contract Access_ArrayElement
{
// Declaring an array and
 // access values from the array
// from a specific index
        uint[6] data;
        uint x;

        function array_example() public
    {
                data= [uint(10), 20, 30, 40, 50, 60];
                 x = data[2];
    }

// Defining function to
function Access_element() public view returns (uint[6] memory,uint)
{
        return (data,x);  }   }
```

Output:


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

```
0: uint256[6]: 10,20,30,40,50,60
1: uint256: 30
```

```
// Solidity program to demonstrate
// accessing elements of an array
pragma solidity ^0.5.0;

// Creating a contract
contract Access_ArrayElement
{

    // Declaring an array and
    // access values from the array
// from a specific index
    uint[6] data;
    uint x;

    function array_example() public
     {

        data= [uint(10), 20, 30, 40, 50, 60];
         x = data[2];
     }

// Defining function to

function Access_element() public view returns (uint[6] memory,uint)
{
        return (data,x);
}
}
```

Push: Push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

Pop: Pop is used when the last element of the array is to be removed in any dynamic array.

Program:

// Solidity program to demonstrate

// Push operation

pragma solidity ^0.5.0;

// Creating a contract

contract Array_Push {

    // Defining the array

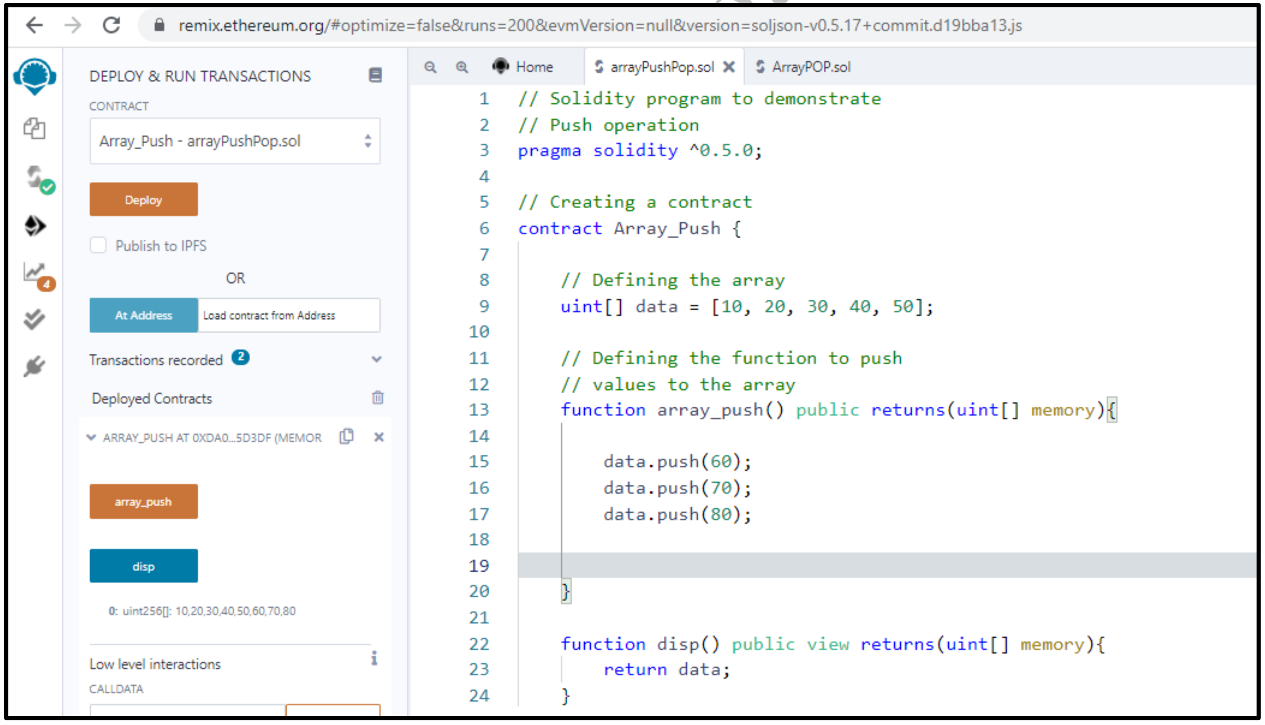
    uint[] data = [10, 20, 30, 40, 50];

    // Defining the function to push

    // values to the array

```
        function array_push() public returns(uint[] memory){
                data.push(60);
                data.push(70);
                data.push(80);
        }

    function disp() public view returns(uint[] memory){
        return data;
        }
}
```

Output:


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

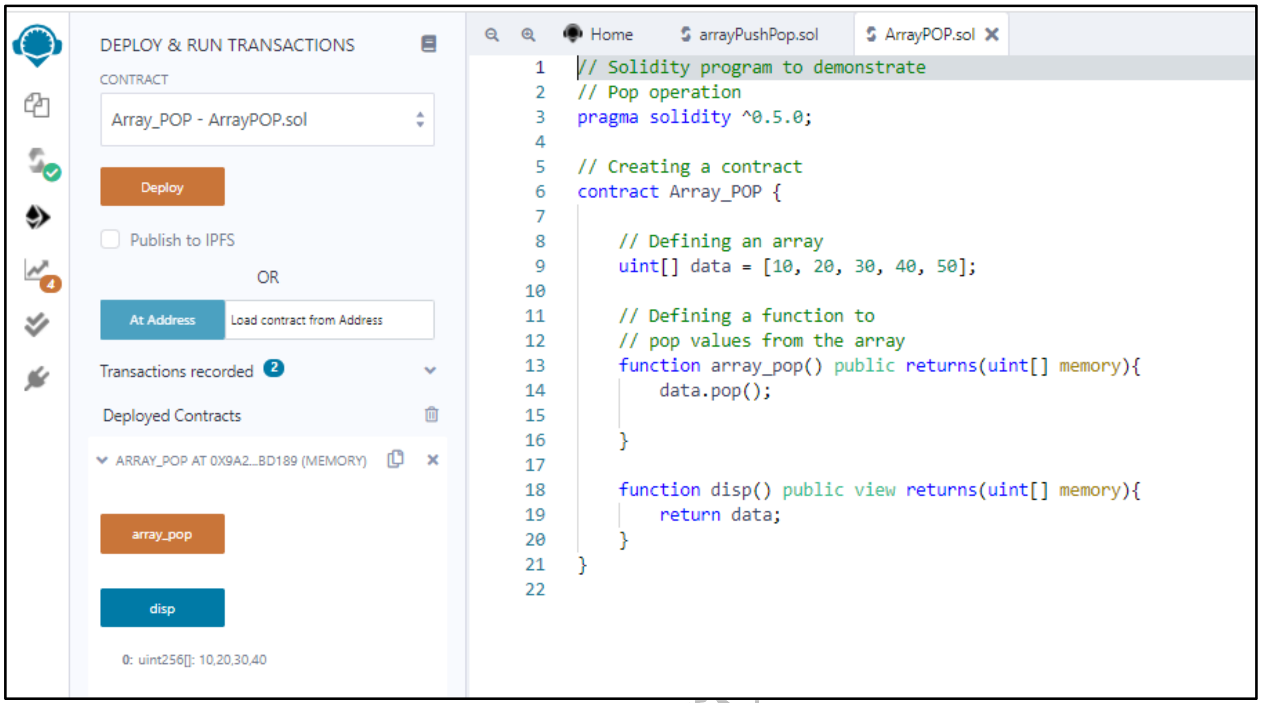
0: uint256[]: 10,20,30,40,50,60,70,80

```
// Solidity program to demonstrate
// Push operation
pragma solidity ^0.5.0;

// Creating a contract
contract Array_Push {

    // Defining the array
    uint[] data = [10, 20, 30, 40, 50];

    // Defining the function to push
    // values to the array
    function array_push() public returns(uint[] memory){

        data.push(60);
        data.push(70);
        data.push(80);

    }

    function disp() public view returns(uint[] memory){
        return data;
    }
```

Program:

```
// Solidity program to demonstrate
// Pop operation
pragma solidity ^0.5.0;

// Creating a contract
contract Array_POP {
        // Defining an array
        uint[] data    = [10, 20, 30, 40, 50];
        // Defining a function to pop values from the array
        function array_pop() public returns(uint[] memory){
                data.pop();
                }
    function disp() public view returns(uint[] memory){
        return data;
        }
}
```

Output:



```
0: uint256[]: 10,20,30,40
```

```
// Solidity program to demonstrate
// Pop operation
pragma solidity ^0.5.0;

// Creating a contract
contract Array_POP {

    // Defining an array
    uint[] data = [10, 20, 30, 40, 50];

    // Defining a function to
    // pop values from the array
    function array_pop() public returns(uint[] memory){
        data.pop();

    }

    function disp() public view returns(uint[] memory){
        return data;
    }
}
```

SOLIDITY – ERROR HANDLING

Solidity has many functions for error handling. Errors can occur at compile time or runtime.

Solidity is compiled to byte code and there a syntax error check happens at compile-time, while runtime errors are difficult to catch and occurs mainly while executing the contracts.

Some of the runtime errors are out-of-gas error, data type overflow error, divide by zero error, array-out-of-index error, etc.

Until version 4.10 a single throw statement was there in solidity to handle errors, so to handle errors multiple if…else statements, one has to implement for checking the values and throw errors which consume more gas.

After version 4.10 new error handling construct assert, require, revert statements were introduced and the throw was made absolute.

Require Statements

The 'require' statements declare prerequisites for running the function i.e. it declares the constraints which should be satisfied before executing the code.
It accepts a single argument and returns a boolean value after evaluation, it also has a custom string message option. If false then exception is raised and execution is terminated.
The unused gas is returned back to the caller and the state is reversed to its original state. Following are some cases when require type of exception is triggered :

When require() is called with the arguments which result as false.
When a function called by a message does not end properly.
When a contract is created using the new keyword and the process does not end properly.
When a codeless contract is targeted to an external function.
When ethers are sent to the contract using the public getter method.
When .transfer() method fails.

When an assert is called with a condition that results in false.

When a zero-initialized variable of a function is called.

When a large or a negative value is converted to an enum.

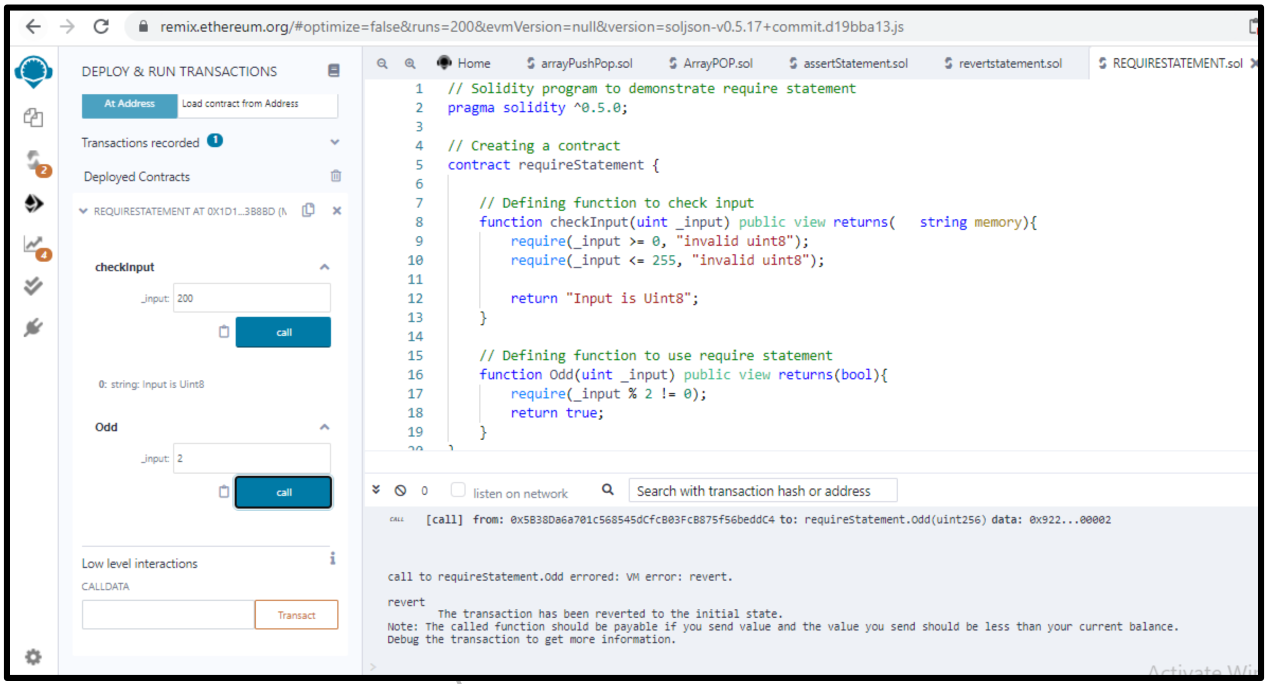
When a value is divided or modulo by zero.

When accessing an array in an index which is too big or negative.

Program:

```
// Solidity program to demonstrate require statement
pragma solidity ^0.5.0;

// Creating a contract
contract requireStatement {

        // Defining function to check input
        function checkInput(uint _input) public view returns( string memory){
                require(_input >= 0, "invalid uint8");
                require(_input <= 255, "invalid uint8");

                return "Input is Uint8";
        }
        // Defining function to use require statement
        function Odd(uint _input) public view returns(bool){
                require(_input % 2 != 0);
                return true;
        }
}
```

Output:



remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

```
// Solidity program to demonstrate require statement
pragma solidity ^0.5.0;

// Creating a contract
contract requireStatement {

    // Defining function to check input
    function checkInput(uint _input) public view returns(   string memory){
        require(_input >= 0, "invalid uint8");
        require(_input <= 255, "invalid uint8");

        return "Input is Uint8";
    }

    // Defining function to use require statement
    function Odd(uint _input) public view returns(bool){
        require(_input % 2 != 0);
        return true;
    }
}
```

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: requireStatement.Odd(uint256) data: 0x922...00002

call to requireStatement.Odd errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
Debug the transaction to get more information.
```

Assert Statement

Its syntax is similar to the require statement. It returns a boolean value after the evaluation of the condition.

Based on the return value either the program will continue its execution or it will throw an exception. Instead of returning the unused gas, the assert statement consumes the entire gas supply and the state is then reversed to the original state.

Assert is used to check the current state and function conditions before the execution of the contract. Below are some cases with assert type exceptions :

When an assert is called with a condition that results in false.

When a zero-initialized variable of a function is called.

When a large or a negative value is converted to an enum.

When a value is divided or modulo by zero.

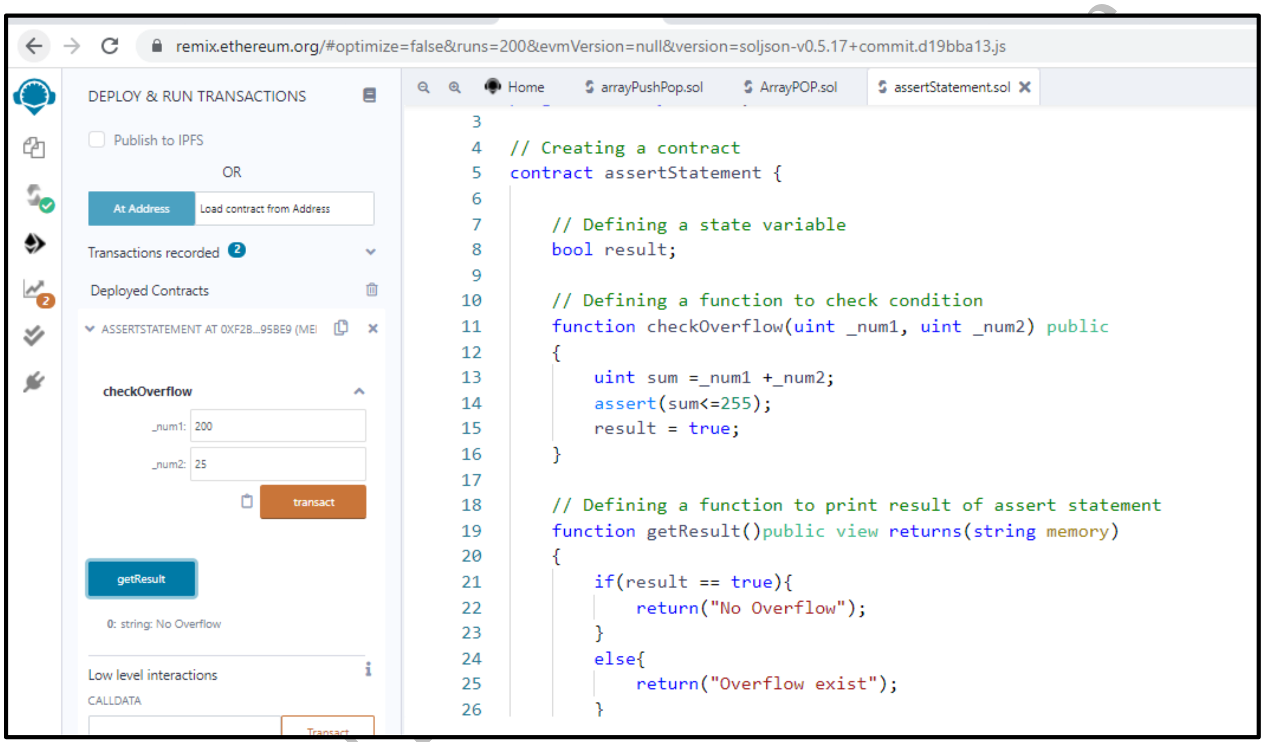
When accessing an array in an index which is too big or negative.

Program:

```
// Solidity program to demonstrate assert statement
pragma solidity ^0.5.0;

// Creating a contract
contract assertStatement {

        // Defining a state variable
        bool result;

        // Defining a function to check condition
        function checkOverflow(uint _num1, uint _num2) public
        {
                uint sum =_num1 +_num2;
                assert(sum<=255);
                result = true;
        }

        // Defining a function to print result of assert statement
        function getResult()public view returns(string memory)
        {
                if(result == true){
                        return("No Overflow");
                }
```

```
                        else{
                                return("Overflow exist");
                        }
                }
        }
```

Output:



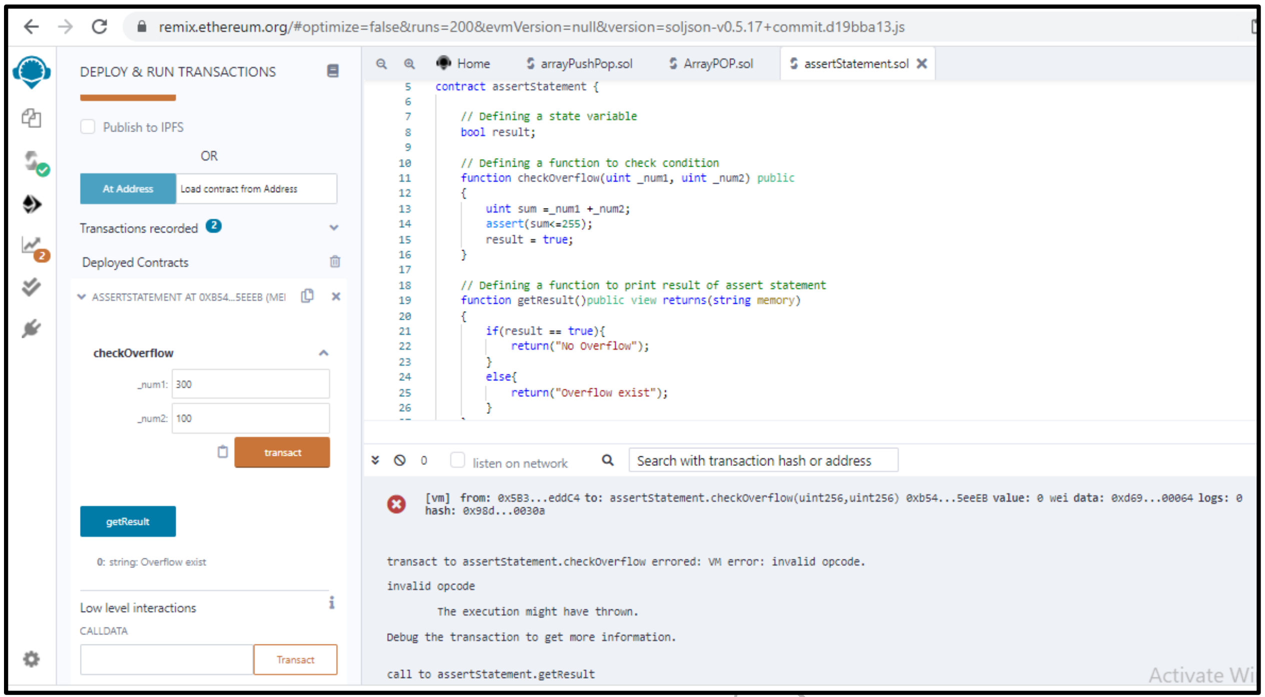
remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

_num1: 200

_num2: 25

0: string: No Overflow

```

// Creating a contract
contract assertStatement {

    // Defining a state variable
    bool result;

    // Defining a function to check condition
    function checkOverflow(uint _num1, uint _num2) public
    {
        uint sum =_num1 +_num2;
        assert(sum<=255);
        result = true;
    }

    // Defining a function to print result of assert statement
    function getResult()public view returns(string memory)
    {
        if(result == true){
            return("No Overflow");
        }
        else{
            return("Overflow exist");
        }
```



remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js

```solidity
contract assertStatement {

    // Defining a state variable
    bool result;

    // Defining a function to check condition
    function checkOverflow(uint _num1, uint _num2) public
    {
        uint sum =_num1 +_num2;
        assert(sum<=255);
        result = true;
    }

    // Defining a function to print result of assert statement
    function getResult()public view returns(string memory)
    {
        if(result == true){
            return("No Overflow");
        }
        else{
            return("Overflow exist");
        }
```

checkOverflow

_num1: 300

_num2: 100

0: string: Overflow exist

```
[vm] from: 0x5B3...eddC4 to: assertStatement.checkOverflow(uint256,uint256) 0xb54...5eeEB value: 0 wei data: 0xd69...00064 logs: 0
hash: 0x98d...0030a

transact to assertStatement.checkOverflow errored: VM error: invalid opcode.
invalid opcode
        The execution might have thrown.
Debug the transaction to get more information.

call to assertStatement.getResult
```

Revert Statement

This statement is similar to the require statement. It does not evaluate any condition and does not depends on any state or statement.

It is used to generate exceptions, display errors, and revert the function call. This statement contains a string message which indicates the issue related to the information of the exception.

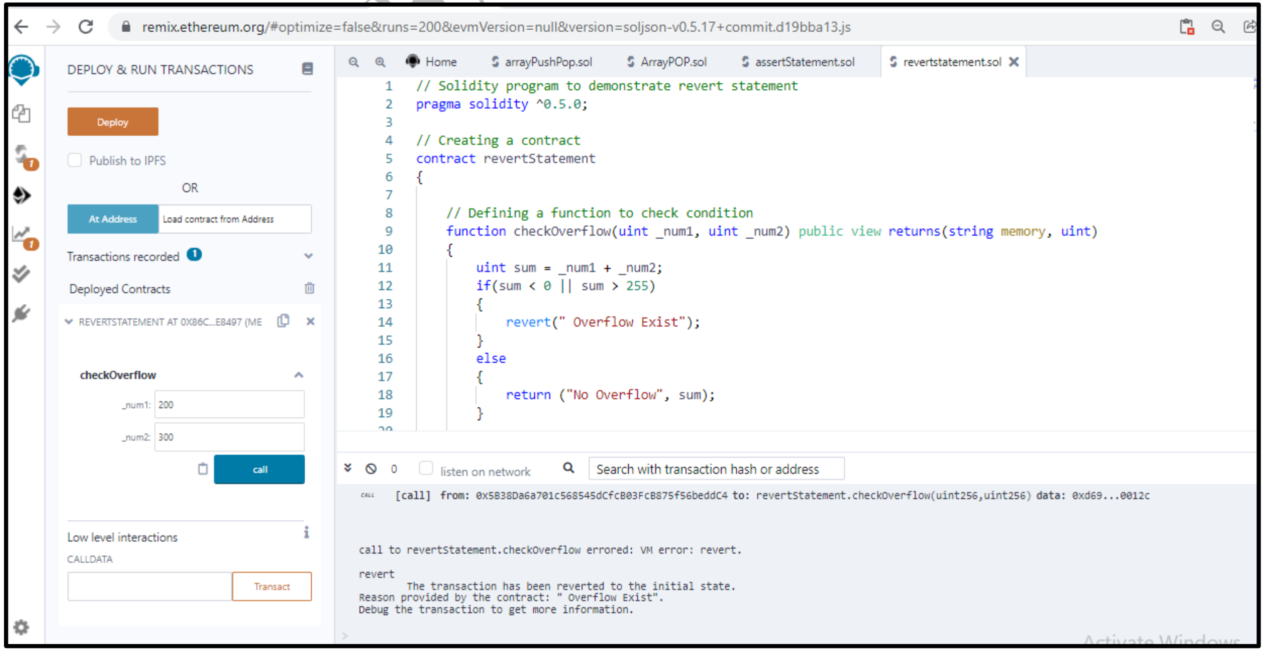
Calling a revert statement implies an exception is thrown, the unused gas is returned and the state reverts to its original state.

Revert is used to handle the same exception types as require handles, but with little bit more complex logic.

Program:

```
// Solidity program to demonstrate revert statement
pragma solidity ^0.5.0;

// Creating a contract
contract revertStatement
{

        // Defining a function to check condition
        function checkOverflow(uint _num1, uint _num2) public view returns(string memory,
uint)
        {
                uint sum = _num1 + _num2;
                if(sum < 0 || sum > 255)
                {
                        revert(" Overflow Exist");
                }
                else
                {
                        return ("No Overflow", sum);
                }

        }

}
```

Output:


remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.5.17+commit.d19bba13.js
DEPLOY & RUN TRANSACTIONS
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 1
Deployed Contracts
REVERTSTATEMENT AT 0X86C...E8497 (ME
checkOverflow
call
Low level interactions
CALLDATA
Transact
Home
arrayPushPop.sol
ArrayPOP.sol
assertStatement.sol
revertstatement.sol
listen on network
Search with transaction hash or address

_num1: 200
_num2: 300

```
// Solidity program to demonstrate revert statement
pragma solidity ^0.5.0;

// Creating a contract
contract revertStatement
{

    // Defining a function to check condition
    function checkOverflow(uint _num1, uint _num2) public view returns(string memory, uint)
    {
        uint sum = _num1 + _num2;
        if(sum < 0 || sum > 255)
        {
            revert(" Overflow Exist");
        }
        else
        {
            return ("No Overflow", sum);
        }
```

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: revertStatement.checkOverflow(uint256,uint256) data: 0xd69...0012c


call to revertStatement.checkOverflow errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: " Overflow Exist".
Debug the transaction to get more information.
```