**PRACTICAL 3**

**3A ) Variable, Operators, Loops, Decision Making, Strings, Arrays, Enums, Structs, Mappings, Conversions, Ether Units, Special Variables.**

**Variable:**

pragmasolidity^0.8.18;

contract Variable {

   uintstoredData;// State variable

   constructor(){

      storedData=40;

   }

   functiongetResult()publicviewreturns(uint){

      uint a =1;// local variable

      uint b =5;

      uint result = a + b;

      require(result >10,"Result must be greater than 10");

      returnstoredData;// access the state variable

   }

}

**Operators :**

**Arthmetic Operator:**

// SPDX-License-Identifier: MIT

pragmasolidity^0.8.18;

contractArithmeticOperators{

    uint16public a =50;

    uint16public b =20;

    uintpublic sum = a + b;

    uintpublic diff = a - b;

    uintpublicmul= a \* b;

    uintpublic div = a / b;

    uintpublic mod = a % b;

    uintpublicdec=--b;

    uintpublicinc=++a;

}

**Relational Operator:**

pragmasolidity^0.8.18;

// Creating a contract

contractRelationalOperator{

    uint16public a =70;

    uint16public b =10;

    boolpubliceq= a == b;

    boolpublicnoteq= a != b;

    boolpublicgtr= a > b;

    boolpublic les = a < b;

    boolpublicgtreq= a >= b;

    boolpublicleseq= a <= b;

}

**Logical Operator:**

pragmasolidity^0.8.18;

// Creating a contract

contractlogicalOperator{

    function Logic(

    bool a,bool b)publicviewreturns(

    bool,bool,bool){

    // Logical AND operator

    bool and = a&&b;

    // Logical OR operator

    bool or = a||b;

    // Logical NOT operator

    bool not =!a;

    return(and, or, not);

}

}

**Bitwise Operator:**

pragmasolidity^0.8.18;

// Creating a contract

contractBitwiseOperator{

    // Declaring variables

    uint16public a =20;

    uint16public b =50;

    uint16public and = a & b;

    uint16public or = a | b;

    uint16publicxor= a ^ b;

    uint16publicleftshift= a << b;

    uint16publicrightshift= a >> b;

    uint16public not =~a;

}

**Assignment Operator:**

pragmasolidity^0.8.18;

// Creating a contract

contractAssignmentOperator{

    // Declaring variables

    uint16public assignment =20;

    uint16publicassignment\_add=50;

    uintpublicassign\_sub=50;

    uintpublicassign\_mul=10;

    uintpublicassign\_div=50;

    uintpublicassign\_mod=32;

    functiongetResult()public{

    assignment\_add+=10;

    assign\_sub-=20;

    assign\_mul\*=10;

    assign\_div/=10;

    assign\_mod%=20;

    return;

    }

}

**Loops**

**whileLoop:**

pragmasolidity^0.8.18;

// Creating a contract

contractwhileLoop{

    //Declaring a dynamic array

    uint[] data;

    //Declaring state variable

    uint8 j =0;

    function loop(

    )publicreturns(uint[]memory){

    while(j <16){

        j++;

        data.push(j);

    }

    return data;

    }

}

**Dowhile loop:**

pragmasolidity^0.8.18;

// Creating a contract

contractdoWhileLoop{

    //Declaring a dynamic array

    uint[] data;

    //Declaring state variable

    uint8 j =0;

    //Defining function to demonstrate

    //'Do-While loop'

    function loop(

    )publicreturns(uint[]memory){

    do{

        j++;

        data.push(j);

    }while(j <8);

    return data;

    }

}

**Forloop:**

pragmasolidity^0.8.18;

//Creating a contract

contractforLoop{

    //Declaring a dynamic array

    uint[] data;

    //Declaring a function

    //to demonstrate 'For loop'

    function loop(

    )publicreturns(uint[]memory){

    for(uint i=0; i<4; i++){

        data.push(i);

    }

    return data;

    }

}

**Decision making**

**If statement:**

pragmasolidity^0.8.18;

//Creating a contract

contractifStatement{

    //Declaring state variable

    uint i =20;

    functiondecision\_making(

    )publicreturns(bool){

        if(i<20){

            returntrue;

        }

    }

}

**If…else statement:**

pragmasolidity^0.8.18;

//Creating a contract

contractifElseStatement{

    //Declaring state variables

    uint i =20;

    bool even;

    //Defining function to

    //Demostrate the use case of

    //'if...else statement'

    functiondecision\_making(

    )publicpayablereturns(bool){

        if(i%2==0){

            even=true;

        }

        else{

            even=false;

        }

        return even;

    }

}

**If…else if…else statement:**

pragmasolidity^0.8.18;

//Creating a contract

contractifElseIfStatement{

    //Declaring state variables

    uint i =12;

    string result;

    functiondecision\_making(

    )publicreturns(stringmemory){

        if(i<10){

            result="less than 10";

        }

        elseif(i ==10){

            result="equal to 10";

        }

        else{

            result="greater than 10";

        }

        return result;

    }

}

**String:**

pragmasolidity^0.8.18;

//Creating a contract

contract String {

    string[]public row;

    functiongetRow()publicviewreturns(string[]memory){

        return row;

    }

    functionpushToRow(stringmemorynewValue)public{

        row.push(newValue);

    }

}

**Array:**

pragmasolidity^0.8.18;

contract test {

    functiontestArray()publicpure{

        uintlen=7;

        uint[]memory a =newuint[](7);

        //bytes is same as byte[]

        bytesmemory b =newbytes(len);

        assert(a.length==7);

        assert(b.length==len);

        //access array varaible

        a[6]=8;

        //test array variable

        assert(a[6]==8);

        //static array

        uint[3]memory c =[uint(1),2,3];

        assert(c.length==3);

    }

}

**Enums:**

pragmasolidity^0.8.18;

contractEnums{

    enumFreshJuiceSize{ SMALL, MEDIUM, LARGE}

    FreshJuiceSize choice;

    FreshJuiceSizeconstantdefaultChoice=FreshJuiceSize.MEDIUM;

    functionsetLarge()public{

        choice=FreshJuiceSize.LARGE;

    }

    functiongetChoice()publicviewreturns(FreshJuiceSize){

        return choice;

    }

    functiongetDefaultChoice()publicpurereturns(uint){

        returnuint(defaultChoice);

    }

}

**Struct:**

pragmasolidity^0.8.18;

contractStruct{

    struct Book {

        string title;

        string author;

        uintbook\_id;

    }

    Book book;

    functionsetBook()public{

        book= Book('Learn Javascript','TP',4);

    }

    functiongetBookId()publicviewreturns(uint){

        returnbook.book\_id;

    }

}

**Mapping:**

pragmasolidity^0.8.18;

contract Mapping {

    struct Student {

        string name;

        string subject;

        uint8 marks;

    }

    // Creating mapping

    mapping(address => Student)public result;

    address[]publicstudent\_result;

    // Function adding values to the mapping

    functionadding\_values()public{

        addressstudentAddress=address(0xDEE7796E89C36BAdd1375076f39D69FafE252);

        Student storage student =result[studentAddress];

        student.name ="John";

        student.subject="Chemistry";

        student.marks=88;

        student\_result.push(studentAddress);

    }

}

**Special Variable:**

pragmasolidity^0.8.18;

contractSpecialVariable{

    //Creating a mapping

    mapping(address =>uint)rollNo;

    functionsetRollNO(uint \_myNumber)public

    {

        rollNo[msg.sender]= \_myNumber;

    }

    //Defining a function to

    //Return the roll no.

    functionwhatIsMyRollNumber()

            publicviewreturns(uint)

    {

        returnrollNo[msg.sender];

    }

}

**3B ) Functions, Function Modifiers, View functions, Pure Functions, Fallback Function, Function Overloading, Mathematical functions, Cryptographic functions.**

**Function:**

pragmasolidity^0.8.18;

contract Function {

    functiongetResult()publicviewreturns(uint product,uint sum){

        uint a =11;// local variable

        uint b =20;

        product= a \* b;

        sum= a + b;

    }

}

**Functions Modifiers:**

pragmasolidity^0.8.18;

contract Owner {

    address owner;

    constructor()public{

        owner=msg.sender;

    }

    modifieronlyOwner{

        require(msg.sender== owner);

        \_;

    }

    modifier costs(uint price){

        if(msg.value>= price){

            \_;

        }

    }

}

contract Register is Owner {

    mapping(address =>bool)registeredAddresses;

    uint price;

    constructor(uintinitialPrice)public{ price =initialPrice;}

    function register()publicpayable costs(price){

        registeredAddresses[msg.sender]=true;

    }

    functionchangePrice(uint \_price)publiconlyOwner{

        price= \_price;

    }

}

**View function:**

pragmasolidity^0.8.18;

contractViewFunction{

    uint num1 =2;

    uint num2 =4;

functiongetResult(

)publicviewreturns(

    uint product,uint sum){

    uint num1 =10;

    uint num2 =16;

    product= num1 + num2;

    sum= num1 + num2;

    }

}

**Pure function:**

pragmasolidity^0.8.18;

contractPureFunction{

functiongetResult(

)publicpurereturns(

    uint product,uint sum){

    uint num1 =2;

    uint num2 =10;

    product= num1 \* num2;

    sum= num1 + num2;

    }

}

**Fallback function:**

pragmasolidity^0.8.18;

contractViewFunction{

    uintpublic x;

    receive()externalpayable{

        x =1;

    }

}

contract Sink {

    fallback()externalpayable{}

}

contract Caller {

    functioncallTest(ViewFunctionviewContract)publicreturns(bool){

        (bool success,)=address(viewContract).call(abi.encodeWithSignature("nonExistingFunction"));

        require(success);

        // viewContract.x is now 1

        addresspayableviewPayable=payable(address(viewContract));

        // Sending ether to ViewFunction contract,

        // the transfer will fail, i.e. this returns false here.

        return(viewPayable.send(2 ether));

    }

    functioncallSink(Sink sink)publicreturns(bool){

        addresspayablesinkPayable=payable(address(sink));

        return(sinkPayable.send(2 ether));

    }

}

**Function Overloading:**

pragmasolidity^0.8.18;

contractFunctionOverloading{

    functiongetSum(uint a,uint b)publicpurereturns(uint){

        return a + b;

    }

    functiongetSum(uint a,uint b,uint c)publicpurereturns(uint){

        return a + b + c;

    }

    functioncallSumWithTwoArguments()publicpurereturns(uint){

        returngetSum(1,2);

    }

    functioncallSumWithThreeArguments()publicpurereturns(uint){

        returngetSum(1,2,3);

    }

}

**Mathematical Function:**

pragmasolidity^0.8.18;

contractMathematicalFunction{

    functioncallAddMod()publicpurereturns(uint){

        returnaddmod(14,15,13);

    }

    functioncallMulMod()publicpurereturns(uint){

        returnmulmod(14,15,13);

    }

}

**Cryptographic function:**

pragmasolidity^0.8.18;

contractCrytographicFunction{

    function callKeccak256()publicpurereturns(bytes32 result){

        returnkeccak256("ABC");

    }

}

PRACTICAL 4

Aim : Implement and demonstrate the use of the following in Solidity :

4A) Withdrawal Pattern, RestrictedAccess.

WithdrawalPattern:

pragmasolidity^0.8.18;

contractwithdrawelContract{

    addresspublic richest;

    uintpublicmostSent;

    mapping(address =>uint)pendingwithdrawals;

    error NotEnoughEther();

    constructor()payable{

        richest =msg.sender;

        mostSent=msg.value;

    }

    functionbecomeRichest()publicpayable{

        if(msg.value<=mostSent)revertNotEnoughEther();

        pendingwithdrawals[richest]+=msg.value;

        richest =msg.sender;

        mostSent=msg.value;

    }

    functionwithdraw()public{

        uint amount =pendingwithdrawals[msg.sender];

        pendingwithdrawals[msg.sender]=0;

        payable(msg.sender).transfer(amount);

    }

}

RestrictedAccess:

pragma solidity ^0.4.21;

contractAccessRestriction{

    addresspublic owner =msg.sender;

    uintpubliclastOwnerChange= now;

    modifieronlyBy(address \_account){

        require(msg.sender== \_account);

        \_;

        }

    modifieronlyAfter(uint \_time){

        require(now >= \_time);

        \_;

        }

    modifiercosts(uint \_amount){

        require(msg.value>= \_amount);

        \_;

        if(msg.value> \_amount){

        msg.sender.transfer(msg.value- \_amount);

        }

     }

    functionchangeOwner(address \_newOwner)publiconlyBy(owner){

        owner = \_newOwner;

    }

    functionbuyContract()publicpayableonlyAfter(lastOwnerChange+4 weeks) costs(1 ether){

        owner =msg.sender;

        lastOwnerChange= now;

     }

    }

4B) Contracts, Inheritance, Constructors, Abstract Contracts, Interfaces.

Contracts:

pragmasolidity0.5.0;

contract C {

//private state variable

uintprivate data;

//public state variable

uintpublic info;

//constructor

constructor()public{

info =10;

}

//private function

functionincrement(uint a)privatepurereturns(uint){return a +1;}

//public function

functionupdateData(uint a)public{ data = a;}

functiongetData()publicviewreturns(uint){return data;}

functioncompute(uint a,uint b)internalpurereturns(uint){return a + b;}

}

//External Contract

contract D {

functionreadData()publicreturns(uint){

C c=newC();

c.updateData(7);

returnc.getData();

}

}

//Derived Contract

contract E is C {

uintprivate result;

C private c;

constructor()public{

c =newC();

}

functiongetComputedResult()public{

result =compute(3,5);

}

functiongetResult()publicviewreturns(uint){return result;}

functiongetData()publicviewreturns(uint){return c.info();}

}

**Inheritance:**

pragmasolidity0.5.0;

contract C {

//private state variable

uintprivate data;

//public state variable

uintpublic info;

//constructor

constructor()public{

info =20;

}

//private function

functionincrement(uint a)privatepurereturns(uint){return a +1;}

//public function

functionupdateData(uint a)public{ data = a;}

functiongetData()publicviewreturns(uint){return data;}

functioncompute(uint a,uint b)internalpurereturns(uint){return a + b;}

}

//Derived Contract

contract E is C {

uintprivate result;

C private c;

constructor()public{

c =newC();

}

functiongetComputedResult()public{

result =compute(3,5);

}

functiongetResult()publicviewreturns(uint){return result;}

functiongetData()publicviewreturns(uint){return c.info();}

}

**Constructors :**

pragmasolidity^0.5.0;

// Creating a contract

contractconstructorExample{

// Declaring state variable.

string str;

// Creating a constructor

// to set value of 'str

constructor()public{

str ="This is Example of Constructor";

}

functiongetValue(

)publicviewreturns(

stringmemory){

return str;

}

}

AbstractContracts:

pragmasolidity^0.5.0;

contractabstractConstructor{

functiongetResult()publicviewreturns(uint);

}

contract Test isabstractConstructor{

functiongetResult()publicviewreturns(uint){

uint a =10;

uint b =17;

uint result = a + b;

return result;

}

}

**Interfaces:**

pragmasolidity^0.5.0;

contract Interface {

functiongetResult()publicviewreturns(uint);

}

contract Test is Interface {

functiongetResult()publicviewreturns(uint){

uint a =11;

uint b =67;

uint result = a + b;return result;

}

}

**4C) LIBRARIES, ASSEMBLY, EVENTS, ERROR HANDLING.**

**LIBRARIES:**

pragmasolidity^0.5.0;

library Search {

functionindexOf(uint[]storage self,uint value)publicviewreturns(uint){

    for(uinti=0;i<self.length;i++)if(self[i]== value)returni;

    returnuint(-1);

}

}

contract Library {

uint[] data;

constructor()public{

data.push(1);

data.push(2);

data.push(3);

data.push(4);

data.push(5);

}

functionisValuePresent()externalviewreturns(uint){

uint value =4;

//search if value is present in the array using Library function

uint index =Search.indexOf(data, value);

return index;

}

}

**ASSEMBLY:**

pragmasolidity^0.4.0;

contract Assembly {

    functionadd(uint a)viewreturns(uint b){

        assembly {

            letc := add(a,16)

            mstore(0x80, c)

            {

                letd := add(sload(c),12)

                // assign the value of 'd' to 'b'

                b := d

                // 'd' is deallocated now

            }

            b := add(b, c)

        }

    }

}

**EVENTS:**

// creating an event

pragmasolidity^0.4.21;

// Creating a contract

contract Events {

// Declaring state variables

uint256public value =0;

// Declaring an event

event Increment (address owner);

// Defining a function for logging event

functiongetValue(uint \_a,uint \_b)public{

    emit Increment (msg.sender);

value = \_a + \_b;

}

}

**ERROR HANDLING:**

pragmasolidity^0.5.0;

contractErrorHandling{

    functioncheckInput(uint \_input)publicviewreturns(stringmemory){

        require(\_input >0,"invalid uint8");

        require(\_input <=255,"invalid uint8");

        return"Input is Uint8";

    }

    // Defining function to use require statement

    functionOdd(uint \_input)publicviewreturns(bool){

        require(\_input %2!=0);

        returntrue;

    }

}