**PRACTICAL NO.:3C**

**AIM: WRITE A SOLIDITY PROGRAM FOR FUNCTION, VIEW FUNCTION, PURE**

**FUNCTION & FALLBACK FUNCTION.**

**A)Function:**

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

pragma solidity ^0.5.0;

contract SolidityTest {

constructor() public{

}

function getResult() public view returns(string memory){

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}

}

**B)View Function:**

View functions ensure that they will not modify the state. A function can be declared as **view**. Getter method are by default view functions.

pragma solidity ^0.5.0;

contract Test {

function getResult() public view returns(uint product, uint sum){

uint a = 1; // local variable

uint b = 2;

product = a \* b;

sum = a + b;

}

}

**C)Pure Function:**

Pure functions ensure that they not read or modify the state. A function can be declared as **pure**. Pure functions can use the revert() and require() functions to revert potential state changes if an error occurs.

pragma solidity ^0.5.0;

contract Test {

function getResult() public pure returns(uint product, uint sum){

uint a = 1;

uint b = 2;

product = a \* b;

sum = a + b;

}

}

**D)Fallback Function:**

Fallback function is a special function available to a contract.

pragma solidity ^0.5.0;

contract Test {

uint public x ;

function() external { x = 1; }

}

contract Sink {

function() external payable { }

}

contract Caller {

function callTest(Test test) public returns (bool) {

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

require(success);

// test.x is now 1

address payable testPayable = address(uint160(address(test)));

// Sending ether to Test contract,

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

return (testPayable.send(2 ether));

}

function callSink(Sink sink) public returns (bool) {

address payable sinkPayable = address(sink);

return (sinkPayable.send(2 ether));

}

}

**PRACTICAL NO.:3B**

**AIM: WRITE A SOLIDITY PROGRAM FOR FUNCTION OVERLOADING, MATHEMATICAL FUNCTION & CRYPTOGRAPHIC FUNCTIONS.**

**Function Overloading:**

The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

pragma solidity ^0.5.0;

contract Test {

function getSum(uint a, uint b) public pure returns(uint){

return a + b;

}

function getSum(uint a, uint b, uint c ) public pure returns(uint){

return a + b + c;

}

function callSumWithTwoArguments() public pure returns(uint){

return getSum(2,2);

}

function callSumWithThreeArguments() public pure returns(uint){

return getSum(1,2,4);

}

}

**Mathematical Function:**

Solidity provides inbuilt mathematical functions as well.

pragma solidity ^0.5.0;

contract Test {

function callAddMod() public pure returns(uint){

return addmod(4, 5, 3);

}

function callMulMod() public pure returns(uint){

return mulmod(4, 5, 3);

}

}

**Cryptographic Function:**

Solidity provides inbuilt cryptographic functions as well.

pragma solidity ^0.5.0;

contract Test {

function callKeccak256() public pure returns(bytes32 result){

return keccak256("ABC");

}

}