LAB 15-08-2023

Error Handling

Solidity provides various error handling mechanisms to handle both compile-time and runtime errors.

1) **Compile-time errors** are checked during the compilation process when Solidity code is converted into bytecode.
2) **Runtime errors** occur while executing contracts and can be challenging to identify. Common runtime errors include out-of-gas errors, data type overflow errors, divide by zero errors, and array out-of-index errors.

These errors can impact the execution of smart contracts and should be carefully handled to ensure contract robustness and integrity.

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

These statements are instrumental in **reducing potential attack** vectors by enforcing checks and constraints. By carefully designing these conditions, we can thwart potential vulnerabilities like integer overflow or underflow, array index violations, and other security concerns.

Understanding the differences and **best use cases** for each error handling method is crucial for maintaining the integrity and efficiency of your blockchain applications.

Let's explore each mechanism in detail:

1 Require Statement:

- The **'require'** statement is a powerful error handling mechanism in Solidity.
- The **'require'** statement supports a boolean condition that must evaluate to true for the transaction to proceed.

Condition Check:

➔ If the condition evaluates to **true**, meaning the condition is satisfied, the contract execution continues as normal, and the subsequent code is executed.

➔ If the condition evaluates to **false,** indicating that the condition is not met, the **'Error Message'** gets printed and the contract execution is halted immediately.

Use Cases:

Require should be used to validate conditions such as:

1) Inputs
2) conditions before execution
3) Return values from calls to other functions

Syntax:

```
require(condition, "Error message");
```

If the specified condition is not met, the 'require' statement triggers a revert, rolling back the transaction.

The associated error message helps users understand the cause of the failure.

Example:

```
require(sum > 0 && sum <= 255, " Overflow Exist");
```

GAS CONSUMPTION:

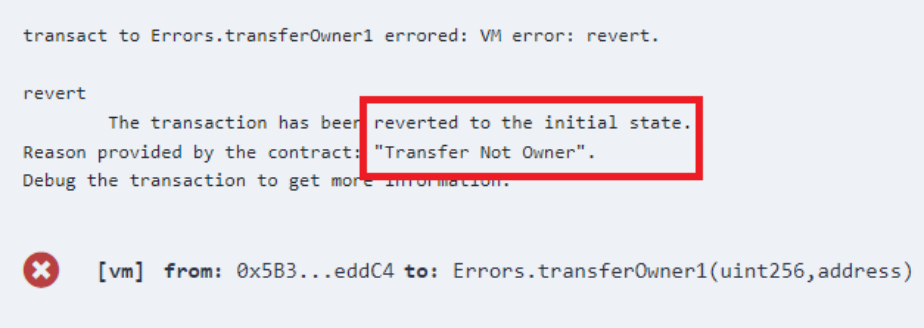
- Despite its simplicity, the gas consumption is usually **higher** than assert and custom error, but quite lesser than revert.
- The gas consumption grows linearly as the length of the error message increases.

Consider the following smart contract,

```
contract OwnerShipTransfer{
    mapping(uint256 => address) private owners;

    function transferOwner1(uint256 ID, address newOwner) public {
        require(owners[ID] == msg.sender, "Transfer Not Owner");
        owners[ID] = newOwner;
    }
```

After deployment, enter a uint256 ‘number’ and a non-zero ‘address’, and call the transferOwner1() function.

The console will throw an error and output the error message **"Transfer Not Owner"**.



And the gas consumption for calling the function, involving the **‘require’** statement is found to be **24711 wei.**

TASK:

Create a Solidity smart contract named **InputValidator**. Inside the contract, define a function named **Transfer** that takes an uint256 ‘amount’, address ‘recipient’.

The function should be capable of transferring the amount from the account of **msg.sender** to the account of the **recipient.**

Use the ‘require’ statement to ensure that the amount is greater than 0 AND less than 100. (0-100).

Hint:

- ➔ Define a mapping named **balances** that maps the address to uint.
- ➔ Initialize balance of **msg.sender** with 10,000 in separate function.

```
balances[msg.sender] = 10000;
```

TASK:

Develop the **‘QuotientCalculator**’ contract with the **‘calculateQuotient**’ function.

Use a **‘require’** statement to ensure the non-zero _denominator before division, reverting with an error if it's zero.

Hint:

To find Quotient, use the **‘/’** operator**.**

2 Revert Statement:

- Unlike **‘require’**, the revert statement does not evaluate any condition and does not depend on any state or statement.
- If a ‘**revert’** statement is called, the unused gas is returned and the state reverts to its original state.
- This statement allows you to explicitly revert the entire transaction state back to its initial state in case of an error.

Use Cases:

➔ Its primary purpose is to ensure that invalid or unexpected states do not persist on the blockchain.

➔ **‘Revert’** is useful when the **condition** to check is **complex.**

Condition Check:

- Since the **‘revert’** statement contains no condition, as soon as execution sequence reaches the revert statement, the Error Message gets displayed and the state reverts to its original state.
- Usually the **‘revert’** statement is controlled by some explicit / outer loop.

Syntax:

```
revert("Error message");
```

Example:

```
if(sum < 0 || sum > 255){
  revert(" Overflow Exist");  }
```

GAS CONSUMPTION:

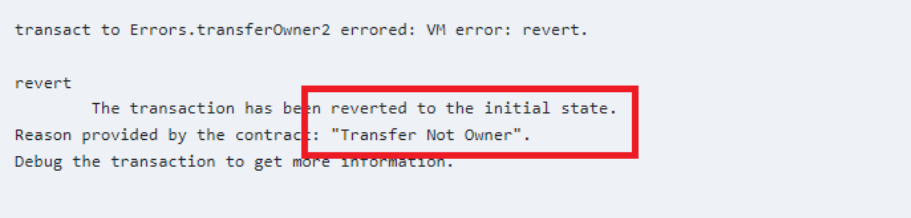
The gas consumption is usually **higher** than other error handling statements.

Consider the following smart contract,

```
contract OwnerShipTransfer{

    mapping(uint256 => address) private owners;


     function transferOwner2(uint256 ID, address newOwner) public {

        if(owners[ID] != msg.sender){

             revert("Transfer Not Owner");     }

        owners[ID] = newOwner;

}
```

After deployment, enter a uint256 ‘number’ and a non-zero ‘address’, and call the transferOwner2() function.

The console will throw an error and output the error message **"Transfer Not Owner"**.



And the gas consumption for calling the function, involving the **‘revert’** statement is found to be **24755 wei.**

TASK

Now, develop **'QuotientCalculator',** but now use '**revert'** statement to ensure the non-zero _denominator before division.

TASK

Design a Solidity contract called **'VotingSystem'**. Implement a function **vote(uint256 _candidateId)** that allows users to vote for a candidate.

Use a '**revert'** statement to handle the case when the user has already voted or when the _candidateId is invalid.

3 Assert Statement

The **assert** statement in solidity is primarily used for debugging purposes. If the condition passed to the **assert** statement is **false**, then the function in which the **assert** is used will revert and any changes made to the state of the contract will be undone.

Use Cases

- ➔ Ensuring critical conditions and contract invariants.
- ➔ Detecting logical errors during development and testing.
- Safeguarding significant state changes.

Syntax

```
assert(condition);
```

Explanation

- **Assert** statements check for conditions that should never be false during program execution.
- If an **assert** statement evaluates to false, it signifies the presence of a bug or an unexpected situation.
- Instead of returning the unused **gas**, the **assert** statement consumes the entire gas supply, and the state is then reversed to the original state.
- Unlike **revert** and **require**, **assert** statements do not provide an error message to users.
- **Assert** should only be used to test for internal errors, and to check invariants.

Example

```
    function sum1(uint8 _num1, uint8 _num2) public pure returns(uint){

        uint sum;
        sum = _num1 + _num2;
        assert(sum < 256);
        return sum;
    }
```

GAS CONSUMPTION:

Assert consumes all gas but lacks user messages.

Consider the following smart contract:

```
contract OwnerShipTransfer{

    mapping(uint256 => address) private owners;

     function transferOwner3(uint256 ID, address newOwner) public {

        assert(owners[ID] == msg.sender);

        owners[ID] = newOwner;
    }
}
```

After deployment, enter a uint256 ‘number’ and a non-zero ‘address’, and call the

transferOwner3() function.

The console will throw an error and any changes made in the function will be reverted.

```
transact to OwnerShipTransfer.transferOwner3 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.
```

[vm] from: 0x5B3...eddC4 **to:** OwnerShipTransfer.transferOwner3(uint256,address) 0xd91...39138 **value:** 0 wei
data: 0x488...39138 **logs:** 0 **hash:** 0x03f...10e4a

And the gas consumption for calling the function, involving the **'assert'** statement is found to be **24450 wei.**

gas	3000000 gas
transaction cost	24450 gas
execution cost	2866 gas

TASK

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

contract Assert{

    function subtract(uint num1, uint num2) public pure returns(uint){

        uint sub;

        sub = num1 - num2;
        return sub;
    }
}
```

- Deploy and observe the code given above.
- Employ an **assert** statement at an appropriate place in the 'subtract()' function to ensure that subtraction result is positive.

Hint:

```solidity
assert(int(num1)-int(num2) >= 0);
```

TASK

Create a **contract** involving a **dynamic array**. Construct a **function** for **adding** elements to the array (**push**), and another function for **removing** elements (**pop**). In the **pop** function, include an **assert** statement to stop users from trying to pop when the array is empty.

TASK

You are assigned a task to create a smart contract that simulates a university admission system. Your goal is to allow students to select their desired academic department only if they have passed their admission.

- Declare an **enum** named **Department** with options: Civil, Mechanic, Electrical, Software.

- Establish two **mappings**: **admissionStatus** to track student admission status (uint => boolean) and **chosenDepartment** to record chosen departments (uint => Department).

- Implement the **setAdmissionStatus** function with parameters:
    - student (student's ID)
    - hasPassed (boolean indicating admission status)

  Use this function to set a student's admission status using **admissionStatus** mapping.

- Implement the **chooseDepartment** function with parameters:
    - student (student's ID)
    - selectedDept (selected 'Department')

- Before assigning the chosen department employ the **assert** statement to verify that the student has met admission requirements (admissionStatus[student] is true).
- If the assertion holds, proceed to assign the chosen department using the **chosenDepartment** mapping.

4 Custom Errors

Solidity allows the creation of **custom errors** to provide specific and meaningful error messages. By defining **custom errors**, you can enhance the clarity and precision of error handling in your contracts thus improving contract usability.

Use Cases

- ➔ Providing user-friendly error messages.
- ➔ Communicating detailed error information.
- ➔ Targeting different error scenarios distinctly.
- ➔ Handling intricate condition checks.
- ➔ Improving code readability.

Declaring Custom Errors

The syntax of **errors** is similar to that of **events**. Custom **errors** can be declared using the **error** keyword, followed by the error name and an optional list of parameters. They can be defined inside and outside of contracts (including interfaces and libraries).

```
error error_name(parameter_list);
```

Reverting Custom Errors

Just like how **events** are **emitted**, custom **errors** have to be used together with the **revert** statement which causes all changes in the current call to be reverted and passes the error data back to the caller.

```
revert error_name(parameter_values);
```

Example:

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

contract CustomError{

    uint public balance = 1000;

    error Balance(uint balance, uint transferAmount);

    function transfer(uint amount) public returns(uint){

        if(amount > balance){

            revert Balance(balance, amount);
        }
        balance -= amount;
        return balance;
```

```
    }
}
```

Key Concepts and Considerations

- By utilizing custom errors, you can provide clear explanations of errors to users and facilitate easier debugging and problem resolution.
- Errors can be inherited, but you cannot override them.
- Solidity compiler version must be greater than 0.8.4 to use custom errors.

GAS CONSUMPTION:

Custom errors require less gas in run time. **Custom errors** also require less memory at contract deployment to store its code, as a result they require less gas at time of deployment.

Consider the following smart contract:

```
contract OwnerShipTransfer{

    error Invalid();

    mapping(uint256 => address) private owners;

     function transferOwner4(uint256 ID, address newOwner) public {

        if(owners[ID] != msg.sender){

             revert Invalid();
            }
        owners[ID] = newOwner;
    }
}
```

After deployment, enter a uint256 ‘number’ and a non-zero ‘address’, and call the transferOwner4() function.

The console will throw an error and any changes made in the function will be reverted.

```
transact to OwnerShipTransfer.transferOwner4 errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Error provided by the contract:
Invalid
Parameters:
{}
Debug the transaction to get more information.
```

[vm] **from:** 0x5B3...eddC4 **to:** OwnerShipTransfer.transferOwner4(uint256,address) 0x358...D5eE3 **value:** 0 wei
data: 0x2a3...d5ee3 **logs:** 0 **hash:** 0x151...97642

And the gas consumption for calling the function is found to be **24458 wei.**

gas	3000000 gas
transaction cost	24458 gas
execution cost	2874 gas

TASK

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

contract CustomError{

    uint[] public arr;


    function upush(uint num) public{

        if(IsPrime(num)){

            //Insert Revert Statement Here
        }

        arr.push(num);
    }

    function IsPrime(uint num) public pure returns(bool){

        if(num == 0 || num == 1){

            return false;
```

```
        }

        for(uint i = 2; i <= num/2; i++){

            if(num % i == 0){

                return true;
            }
        }
        return false;
    }
}
```

- Observe and deploy the above contract.
- Make a **custom error**, having parameters ‘message’ (**string** data type) and ‘num’(**uint** data type).
- Insert a **revert** statement with the **custom error** you have just made in the ‘upush()’ function. The **error** should return a message indicating that the number is not prime along with the number itself.

TASK

- Create a `DriverLicense` **contract** with a **custom error**, `TooYoung`, consisting of a relevant error message (`message` of **string**) and a minimum age requirement (`RequiredAge` of **uint**).
- Implement the `DriverID()` **function**, which takes `age` as **input** and **returns** a driverID (any **uint** number of your choice), if the age is equal to or greater than 18. Otherwise, triggers the `TooYoung` **custom error**.

TASK

Design a smart contract named `ATM` that simulates an Automated Teller Machine (ATM). Your objective is to enhance the contract's error handling mechanism by creating a custom error and integrating a `revert` statement at the appropriate location.

- Initialize a ‘CashAvailable’ **state variable** of type **uint** with an initial value, representing available ATM cash.
- Define a **custom error** named `InsufficientFunds` having a relevant error message (**string** type) along with the current cash available in the ATM (**uint** type).
- Implement a ‘MoneyWithdraw()’ function that:
    - Accepts a ‘WithdrawAmount’ parameter.

    - Uses an **if** statement to check if the ‘CashAvailable’ is less than the ‘WithdrawAmount’.
    - If the condition is met (i.e., insufficient funds), use the **revert** statement to trigger the **custom error** ‘InsufficientFund’. Include the error message and the ‘CashAvailable’ balance in the revert statement.
    - If sufficient funds are available, deduct the ‘WithdrawAmount’ from the ‘CashAvailable’.
- Implement a ‘ATMRefill’ function that:
    - Accept a ‘RefillAmount’ parameter.
    - Add the ‘RefillAmount’ to the current ‘CashAvailable’ balance.