**Эти задачи помогут тебе глубже понять основы Solidity и практиковаться в написании смарт-контрактов.**

**Уровень 1: Введение в Solidity**

**Задача 1: Простой контракт хранения числа.** Напиши контракт, который будет хранить одно число и позволять его обновлять.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для хранения числа

contract SimpleStorage {

// Переменная для хранения числа

uint256 public storedData;

// Функция для обновления числа

function set(uint256 x) public {

storedData = x;

}

// Функция для получения числа

function get() public view returns (uint256) {

return storedData;

}

}

**Задача 2: Контракт для хранения строки** Напиши контракт, который будет хранить строку и позволять её обновлять.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для хранения строки

contract StringStorage {

// Переменная для хранения строки

string public storedString;

// Функция для обновления строки

function set(string memory newString) public {

storedString = newString;

}

// Функция для получения строки

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

return storedString;

}

}

**Задача 3: Контракт с булевым значением** Создай контракт, который будет хранить булевое значение и позволять его обновлять.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для хранения булевого значения

contract BooleanStorage {

// Переменная для хранения булевого значения

bool public storedBoolean;

// Функция для обновления булевого значения

function set(bool x) public {

storedBoolean = x;

}

// Функция для получения булевого значения

function get() public view returns (bool) {

return storedBoolean;

}

}

**Задача 4: Контракт с массивом чисел** Напиши контракт, который будет хранить массив чисел и позволять добавлять новые числа в массив.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для хранения массива чисел

contract ArrayStorage {

// Массив для хранения чисел

uint256[] public numbers;

// Функция для добавления числа в массив

function addNumber(uint256 number) public {

numbers.push(number);

}

// Функция для получения числа из массива по индексу

function getNumber(uint256 index) public view returns (uint256) {

require(index < numbers.length, "Index out of bounds");

return numbers[index];

}

}

**Задача 5: Контракт со словарём адресов** Создай контракт, который будет хранить словарь адресов и позволять добавлять новые пары "адрес - значение".

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для хранения словаря адресов и значений

contract AddressMap {

// Словарь для хранения адресов и значений

mapping(address => uint256) public balances;

// Функция для установки значения для адреса

function setBalance(address addr, uint256 balance) public {

balances[addr] = balance;

}

// Функция для получения значения по адресу

function getBalance(address addr) public view returns (uint256) {

return balances[addr];

}

}

**Уровень 2: Основы Solidity**

**Задача 6: Контракт с управляющим** Напиши контракт, который будет содержать переменную "владелец" и позволять только владельцу выполнять определённые функции.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для управления доступом на основе владельца

contract OwnerContract {

// Переменная для хранения адреса владельца

address public owner;

// Конструктор, который устанавливает владельца контракта

constructor() {

owner = msg.sender;

}

// Модификатор для ограничения доступа только владельцем

modifier onlyOwner() {

require(msg.sender == owner, "Not the owner");

\_;

}

// Функция для изменения владельца, доступная только текущему владельцу

function changeOwner(address newOwner) public onlyOwner {

owner = newOwner;

}

}

**Задача 7: Контракт для вычисления суммы** Создай контракт, который будет вычислять сумму двух чисел, переданных в функцию.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для вычисления суммы двух чисел

contract Adder {

// Функция для вычисления суммы двух чисел

function add(uint256 a, uint256 b) public pure returns (uint256) {

return a + b;

}

}

**Задача 8: Контракт для вычисления разности** Создай контракт, который будет вычислять разность двух чисел, переданных в функцию.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для вычисления разности двух чисел

contract Subtracter {

// Функция для вычисления разности двух чисел

function subtract(uint256 a, uint256 b) public pure returns (uint256) {

return a - b;

}

}

**Задача 9: Контракт для умножения** Напиши контракт, который будет вычислять произведение двух чисел, переданных в функцию.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для вычисления произведения двух чисел

contract Multiplier {

// Функция для вычисления произведения двух чисел

function multiply(uint256 a, uint256 b) public pure returns (uint256) {

return a \* b;

}

}

**Задача 10: Контракт для деления** Напиши контракт, который будет вычислять частное двух чисел, переданных в функцию. Убедись, что делитель не равен нулю.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для вычисления частного двух чисел

contract Divider {

// Функция для вычисления частного двух чисел

function divide(uint256 a, uint256 b) public pure returns (uint256) {

require(b > 0, "Division by zero");

return a / b;

}

}

**Уровень 3: Модификаторы и условия**

**Задача 11: Контракт с условным увеличением** Напиши контракт, который будет увеличивать значение на 1, только если текущее значение меньше заданного порога.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract IncrementWithThreshold {

// Текущее значение

uint256 private value;

// Пороговое значение

uint256 public threshold;

// Событие для логирования увеличения значения

event ValueIncremented(uint256 newValue);

// Конструктор, который устанавливает начальное значение и порог

constructor(uint256 initialValue, uint256 initialThreshold) {

value = initialValue;

threshold = initialThreshold;

}

// Функция для получения текущего значения

function getValue() public view returns (uint256) {

return value;

}

// Функция для увеличения значения на 1, если оно меньше порога

function increment() public {

require(value < threshold, "Value has reached the threshold");

value += 1;

// Логируем увеличение значения

emit ValueIncremented(value);

}

// Функция для установки нового порога

function setThreshold(uint256 newThreshold) public {

threshold = newThreshold;

}

}

**Задача 12: Контракт с ограниченным доступом** Создай контракт, который будет позволять выполнять функцию только пользователю с определённым адресом.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для ограничения доступа по адресу

contract RestrictedAccess {

// Переменная для хранения разрешённого адреса

address public allowedAddress;

// Конструктор для установки разрешённого адреса

constructor(address \_allowedAddress) {

allowedAddress = \_allowedAddress;

}

// Модификатор для ограничения доступа только разрешённым адресом

modifier onlyAllowed() {

require(msg.sender == allowedAddress, "Access denied");

\_;

}

// Функция, доступная только разрешённому адресу

function restrictedFunction() public onlyAllowed {

// Действия, доступные только allowedAddress

}

}

**Задача 13: Контракт с временной блокировкой** Создай контракт, который будет блокировать выполнение функции на определённый период времени после её вызова.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для временной блокировки выполнения функции

contract TimedLock {

// Переменная для хранения времени последнего вызова функции

uint256 public lastExecuted;

// Период блокировки

uint256 public lockTime;

// Конструктор для установки периода блокировки

constructor(uint256 \_lockTime) {

lockTime = \_lockTime;

}

// Функция, которая может быть выполнена только после истечения периода блокировки

function execute() public {

require(block.timestamp >= lastExecuted + lockTime, "Function is locked");

lastExecuted = block.timestamp;

// Действия функции

}

}

**Задача 14: Контракт с условным выполнением** Создай контракт, который будет выполнять функцию только если переданное значение удовлетворяет определённым условиям.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ConditionChecker {

// Событие для логирования успешного выполнения функции

event FunctionExecuted(address indexed caller, uint256 value);

// Функция, которая выполняется только при выполнении условия

function executeIfPositive(uint256 value) public {

// Проверка условия: значение должно быть больше нуля

require(value > 0, "Value must be greater than zero");

// Логирование успешного выполнения функции

emit FunctionExecuted(msg.sender, value);

// Здесь может быть логика, которая выполняется при выполнении условия

// Например, обновление состояния контракта или вызов других функций

}

}

**Задача 15: Контракт с проверкой входных данных** Напиши контракт, который будет проверять входные данные перед выполнением функции.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для проверки входных данных

contract InputValidation {

// Функция для проверки и обработки входных данных

function validate(uint256 value) public pure returns (string memory) {

require(value != 0, "Value must be non-zero");

return "Valid input";

}

}

**Уровень 4: Работа с массивами и маппингами**

**Задача 16: Контракт с удалением элементов массива** Напиши контракт, который будет позволять удалять элементы из массива по индексу.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для удаления элементов массива по индексу

contract ArrayDeletion {

// Массив для хранения чисел

uint256[] public numbers;

// Функция для добавления числа в массив

function addNumber(uint256 number) public {

numbers.push(number);

}

// Функция для удаления числа из массива по индексу

function removeNumber(uint256 index) public {

require(index < numbers.length, "Index out of bounds");

for (uint256 i = index; i < numbers.length - 1; i++) {

numbers[i] = numbers[i + 1];

}

numbers.pop();

}

// Функция для получения числа из массива по индексу

function getNumber(uint256 index) public view returns (uint256) {

require(index < numbers.length, "Index out of bounds");

return numbers[index];

}

}

**Задача 17: Контракт с подсчётом элементов массива** Создай контракт, который будет возвращать количество элементов в массиве.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для подсчёта элементов массива

contract ArrayCount {

// Массив для хранения чисел

uint256[] public numbers;

// Функция для добавления числа в массив

function addNumber(uint256 number) public {

numbers.push(number);

}

// Функция для получения количества элементов в массиве

function getCount() public view returns (uint256) {

return numbers.length;

}

}

**Задача 18: Контракт с итерацией по массиву** Напиши контракт, который будет суммировать все элементы в массиве.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для суммирования всех элементов массива

contract ArraySum {

// Массив для хранения чисел

uint256[] public numbers;

// Функция для добавления числа в массив

function addNumber(uint256 number) public {

numbers.push(number);

}

// Функция для суммирования всех чисел в массиве

function getSum() public view returns (uint256) {

uint256 sum = 0;

for (uint256 i = 0; i < numbers.length; i++) {

sum += numbers[i];

}

return sum;

}

}

**Задача 19: Контракт с проверкой существования ключа в маппинге** Создай контракт, который будет проверять, существует ли ключ в маппинге.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract KeyExistence {

// Объявляем маппинг

mapping(address => uint256) public balances;

// Объявляем дополнительный маппинг для отслеживания существующих ключей

mapping(address => bool) private keyExists;

// Функция для добавления данных в маппинг

function addBalance(address \_address, uint256 \_balance) public {

balances[\_address] = \_balance;

keyExists[\_address] = true;

}

// Функция для проверки существования ключа

function exists(address \_address) public view returns (bool) {

return keyExists[\_address];

}

}

**Задача 20: Контракт с циклическим массивом** Напиши контракт, который будет использовать циклический массив для хранения значений.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CircularBuffer {

// Массив для хранения значений

uint256[] private buffer;

// Максимальный размер буфера

uint256 private size;

// Текущий индекс для добавления нового значения

uint256 private index;

// Конструктор, который инициализирует размер буфера и массив

constructor(uint256 \_size) {

require(\_size > 0, "Size must be > 0");

buffer = new uint256[](\_size);

size = \_size;

index = 0;

}

// Функция для добавления нового значения в буфер

function add(uint256 value) public {

buffer[index] = value;

// Обновляем индекс, используя модульный оператор для циклического перемещения по массиву

index = (index + 1) % size;

}

// Функция для получения значения по индексу

function get(uint256 idx) public view returns (uint256) {

require(idx < size, "Index out of bounds");

// Используем модульный оператор для корректного доступа к значениям в циклическом массиве

return buffer[(index + idx) % size];

}

// Функция для получения текущего размера буфера

function getSize() public view returns (uint256) {

return size;

}

}

**Уровень 5: Продвинутые функции и события**

**Задача 21: Контракт с событием при обновлении данных** Создай контракт, который будет генерировать событие при обновлении данных.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DataStorage {

// Переменная для хранения данных

string private data;

// Определяем событие, которое будет срабатывать при обновлении данных

event DataUpdated(string oldData, string newData);

// Функция для установки данных

function setData(string memory newData) public {

// Храним старые данные для события

string memory oldData = data;

// Обновляем данные

data = newData;

// Генерируем событие при обновлении данных

emit DataUpdated(oldData, newData);

}

// Функция для получения текущих данных

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

return data;

}

}

**Задача 22: Контракт с событием при добавлении элемента в массив** Напиши контракт, который будет генерировать событие при добавлении нового элемента в массив.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ArrayContract {

// Объявление массива

uint[] public array;

// Объявление события

event ElementAdded(uint indexed element);

// Функция для добавления нового элемента в массив

function addElement(uint \_element) public {

// Добавление элемента в массив

array.push(\_element);

// Генерация события ElementAdded

emit ElementAdded(\_element);

}

// Функция для получения всех элементов массива

function getArray() public view returns (uint[] memory) {

return array;

}

}

**Задача 23: Контракт с ограничением количества вызовов функции** Создай контракт, который будет ограничивать количество вызовов функции до заданного значения.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CallLimiter {

address public owner; // Адрес владельца контракта

uint public maxCalls; // Максимальное количество вызовов функции

uint public callCount; // Текущее количество вызовов функции

uint public rewardAmount; // Сумма вознаграждения за каждый вызов

// Событие, которое вызывается при каждом успешном вызове функции

event FunctionCalled(address indexed caller, uint callNumber, uint reward);

// Модификатор для проверки, что функцию вызывает владелец контракта

modifier onlyOwner() {

require(msg.sender == owner, "Not the contract owner");

\_;

}

// Модификатор для проверки, что текущее количество вызовов меньше установленного лимита

modifier callLimit() {

require(callCount < maxCalls, "Call limit exceeded");

\_;

}

// Конструктор, который устанавливает владельца, максимальное количество вызовов и сумму вознаграждения

constructor(uint \_maxCalls, uint \_rewardAmount) payable {

// Проверка, что отправлено достаточно ETH для всех вознаграждений

require(msg.value >= \_rewardAmount \* \_maxCalls, "Not enough ether sent");

owner = msg.sender; // Установка владельца контракта

maxCalls = \_maxCalls; // Установка максимального количества вызовов

rewardAmount = \_rewardAmount; // Установка суммы вознаграждения

callCount = 0; // Инициализация счетчика вызовов

}

// Функция, которая может быть вызвана только ограниченное количество раз

function limitedFunction() public callLimit {

callCount++; // Увеличение счетчика вызовов

payable(msg.sender).transfer(rewardAmount); // Перевод вознаграждения на адрес вызывающего

emit FunctionCalled(msg.sender, callCount, rewardAmount); // Вызов события

}

// Функция для сброса счетчика вызовов (доступна только владельцу)

function resetCounter() public onlyOwner {

callCount = 0; // Сброс счетчика вызовов

}

// Функция для обновления максимального количества вызовов (доступна только владельцу)

function updateMaxCalls(uint \_maxCalls) public onlyOwner {

// Проверка, что в контракте достаточно средств для новых вызовов

require(address(this).balance >= rewardAmount \* \_maxCalls, "Not enough ether in contract");

maxCalls = \_maxCalls; // Обновление максимального количества вызовов

}

// Функция для обновления суммы вознаграждения (доступна только владельцу)

function updateRewardAmount(uint \_rewardAmount) public onlyOwner {

// Проверка, что в контракте достаточно средств для новых вознаграждений

require(address(this).balance >= \_rewardAmount \* maxCalls, "Not enough ether in contract");

rewardAmount = \_rewardAmount; // Обновление суммы вознаграждения

}

// Функция для депонирования дополнительных средств в контракт (доступна только владельцу)

function deposit() public payable onlyOwner {}

// Функция для вывода всех средств из контракта (доступна только владельцу)

function withdraw() public onlyOwner {

payable(owner).transfer(address(this).balance); // Перевод всех средств на адрес владельца

}

}

**Задача 24: Контракт с условной проверкой параметров** Напиши контракт, который будет проверять, что переданные в функцию параметры удовлетворяют заданным условиям.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract Validator {

// Событие для уведомления о проверке параметров

event ParametersValidated(address indexed \_address, uint256 \_number, string \_text);

// Функция для проверки параметров

function validateParameters(address \_address, uint256 \_number, string memory \_text) public {

// Проверка, что адрес не нулевой

require(\_address != address(0), "Address cannot be zero");

// Проверка, что число больше нуля

require(\_number > 0, "Number must be greater than zero");

// Проверка, что строка не пустая

require(bytes(\_text).length > 0, "Text cannot be empty");

// Если все проверки пройдены, генерируем событие

emit ParametersValidated(\_address, \_number, \_text);

}

}

**Задача 25: Контракт с возвратом нескольких значений** Напиши контракт, который будет возвращать несколько значений из функции.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MultiReturnExample {

// Функция, которая возвращает несколько значений

function getValues() public pure returns (uint256, bool, string memory) {

uint256 number = 42;

bool isTrue = true;

string memory message = "Hello, Solidity!";

return (number, isTrue, message);

}

}

**Задача 26: Контракт с временной меткой** Создай контракт, который будет сохранять временную метку последнего вызова функции.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract TimestampRecorder {

// Переменная для хранения временной метки последнего вызова функции

uint256 public lastUpdated;

// Функция для обновления временной метки

function updateTimestamp() public {

lastUpdated = block.timestamp; // Сохраняем текущую временную метку

}

// Функция для получения текущей временной метки

function getLastUpdated() public view returns (uint256) {

return lastUpdated;

}

}

**Задача 27: Контракт с проверкой роли** Напиши контракт, который будет проверять роль пользователя перед выполнением функции.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Импортируем контракт AccessControl из библиотеки OpenZeppelin

import "@openzeppelin/contracts/access/AccessControl.sol";

// Определяем контракт RoleBasedAccessControl, который наследует функционал управления ролями из AccessControl

contract RoleBasedAccessControl is AccessControl {

// Константы для определения ролей с использованием хеш-функции keccak256

bytes32 public constant ADMIN\_ROLE = keccak256("ADMIN\_ROLE");

bytes32 public constant USER\_ROLE = keccak256("USER\_ROLE");

// Конструктор контракта, который вызывается при развертывании контракта

constructor(address initialAdmin) {

// Назначаем ADMIN\_ROLE указанному адресу (администратору)

\_grantRole(ADMIN\_ROLE, initialAdmin);

// Назначаем USER\_ROLE указанному адресу (пользователю)

\_grantRole(USER\_ROLE, initialAdmin);

}

// Функция, доступная только пользователям с ролью ADMIN\_ROLE

function adminFunction() public onlyRole(ADMIN\_ROLE) {

// Логика, доступная только администраторам

}

// Функция, доступная только пользователям с ролью USER\_ROLE

function userFunction() public onlyRole(USER\_ROLE) {

// Логика, доступная только пользователям

}

// Функция для добавления нового пользователя (назначение роли USER\_ROLE)

function addUser(address account) public onlyRole(ADMIN\_ROLE) {

// Назначаем USER\_ROLE указанному адресу

grantRole(USER\_ROLE, account);

}

// Функция для удаления пользователя (удаление роли USER\_ROLE)

function removeUser(address account) public onlyRole(ADMIN\_ROLE) {

// Удаляем роль USER\_ROLE у указанного адреса

revokeRole(USER\_ROLE, account);

}

}

Задача 28: Контракт с уникальными значениями в массиве Создай контракт, который будет добавлять в массив только уникальные значения.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для добавления в массив только уникальных значений

contract UniqueArray {

// Массив для хранения уникальных значений

uint256[] public uniqueValues;

// Событие, которое будет эмитироваться при добавлении нового значения

event ValueAdded(uint \_value);

// Функция для добавления уникального значения в массив

function addUniqueValue(uint256 \_value) public {

for (uint256 i = 0; i < uniqueValues.length; i++) {

require(uniqueValues[i] != \_value, "Value already exists");

}

uniqueValues.push(\_value);

// Эмитируем событие добавления значения

emit ValueAdded(\_value);

}

// Функция для получения значения из массива по индексу

function getValue(uint256 index) public view returns (uint256) {

require(index < uniqueValues.length, "Index out of bounds");

return uniqueValues[index];

}

// Функция для получения всех уникальных значений

function getValues() public view returns (uint[] memory) {

return uniqueValues;

}

}

**Задача 29: Контракт с подсчётом уникальных значений** Напиши контракт, который будет подсчитывать количество уникальных значений в массиве.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для подсчёта количества уникальных значений в массиве

contract UniqueCount {

// Массив для хранения уникальных значений

uint256[] public uniqueValues;

// Функция для добавления уникального значения в массив

function addUniqueValue(uint256 value) public {

for (uint256 i = 0; i < uniqueValues.length; i++) {

require(uniqueValues[i] != value, "Value already exists");

}

uniqueValues.push(value);

}

// Функция для получения количества уникальных значений в массиве

function getUniqueCount() public view returns (uint256) {

return uniqueValues.length;

}

}

**Задача 30: Контракт с обратной связью** Создай контракт, который будет генерировать событие с обратной связью при выполнении функции.

solidity

Копировать код

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Контракт для генерации события с обратной связью

contract FeedbackContract {

// Определяем событие, которое будет содержать адрес вызывающего, сообщение и статус выполнения

event FeedbackGenerated(address indexed user, string message, bool success);

// Пример функции, которая вызывает событие при своем выполнении

function performAction(string memory \_input) public returns (bool) {

bool success = false;

// Логика выполнения действия

if (bytes(\_input).length > 0) {

success = true;

}

// Генерация события с обратной связью

emit FeedbackGenerated(msg.sender, "Action has been performed", success);

return success;

}

}