TIF5211 - Special Topics for Information Technology

Introduction to Smart Contract Programming

gdputra@ugm.ac.id

Courtesy: UNSW COMP6452

Introduction to Smart Contract Programming

Hands-on sub-session

We will write a smart contract, then deploy it, and test it. You should follow the documentation of your compiler version. There is a cheatsheet for your reference. When you are unsure about some features (this is very common as features are changing from version to version), you may also figure out through experiments.

The smart contract is compiled into `opcodes` of bytecode form and generates an `ABI`. The bytecodes are deployed onto the blockchain and executed by Ethereum Virtual Machine (EVM).

Remix is an official online IDE of Ethereum. It supports `Solidity` and `Vyper`.

Workspace structure

A typical Remix workspace has the following structure:

```
contracts/
    SomeContract.sol
    ...
tests/
    SomeContract_test.sol
    ...
```

Contract and its members

The contract in Solidity is like a class. It may contain variables and functions with different levels of visibility. The constructor may also take in parameters. Functions may return multiple values.

```
contract A {
    int x;
    bool private y;

    constructor() {

    }

    function z(int input) public returns (bool) {
        (x, y) = twoRet();
        return false;
    }

    function twoRet() public returns (int, bool) {
        return (1, true);
    }
}
```

Types

Supported types in Solidity are limited compared with other programming languages, e.g. no floating point types. Operators are similar to others.

```
int a;
uint b;
uint8 c;
int16 d;
...
bool e;
address f;
mapping(string=>int) g;
```

Arrays, strings, bytes, and structs are reference types which are a bit special.

There are two keywords to specify the data location: `storage` and `memory`. Data defined in the contract domain are stored in storage, and function-domain variables are references. These references can be references to storage variables, or memory variables. Where to store these data is determined by their lifespan. Temporary data are usually stored in memory.

Arrays defined in the contract domain can be of dynamic size.

```
contract A {
    int[] b;

    function c() public {
        b.push(1);
        b[0] = 2;
        b[99] = 10; // will be reverted if b.length < 100
        b.pop();
    }
}
```

Arrays defined in the function domain are reference variables. **The initialization of a memory array must have fixed size and cannot be resized.**

```
contract A {
    function f(int len) {
        int[] memory f = new int[](len);
    }
}
```

Defining a struct is similar to other languages, but use a semicolon after each field instead of the comma.

```
struct Car {
    string brand;
    uint numberOfSeats;
}

struct CarPark {
    Car[] cars;
}
```

Arrays and mappings can also be nested.

```
int[2][] arr;

mapping(string=>mapping(string=>bool)) nestedMap;
```

Enumeration

Enumeration is useful for indicating states. It can be defined in a contract or in a file domain.

```
enum States { StateA, StateB, StateC, StateD }

...
    if(state == States.StateA) {
        ...
    }else if (state == States.StateB) {
        ...
    }
...
```

Function declarations and modifiers

Functions can be declared to indicate its nature of idempotence. Using `pure` keyword declares that the function will not read or modify the states, and using `view` keyword declares that the function will not modify the states. With warning toggled on, the compiler will automatically raise the warnings if it finds the function should be `pure` or `view`.

```
contract Example {
    int aNumber;
    function aPureFunction(int a, int b) public pure returns (int) {
        return a + b;
    }

    function aViewFunction(int a, int b) public view returns (int) {
        return aNumber + a + b;
    }
}
```

You can also enforce requirements for functions by using `require` keyword, which is usually done in modifier functions. The most used modifier is the access control. Modifiers may also receive parameters, which is not mentioned in the documentation.

```
contract Example {
    address private owner;

    constructor() {
        owner = msg.sender;
    }

    modifier isOwner() {
        require(msg.sender == owner);
        _;
    }

    function changeCriticalState() public isOwner {
        ...
    }

    modifier onlyHappy(bool isHappy) {
        require(isHappy == true);
        _;
    }

    function happyCaller(bool isHappy) public onlyHappy(isHappy) {
        ...
    }
}
```

Special variables

You may find `msg.sender` and `block.number` very useful for project 1. The former one gives the sender's address, and the latter one gives the current number of blocks. However, the address returned by `msg.sender` can sometime be confusing. It returns the caller contract address if the function is called by another contract, and in this case if you want to get the caller account address of the whole invocation chain, you should use `tx.origin`.

A full list of special variables can be found in the documentation.

Interactions between smart contracts

Calling a function in another contract must specify the exact contract.

```
contract A {
    function invokeBF(address b) public {
        B(b).f();
    }
}

contract B {
    function f() public {

    }
}
```

You may deploy new contract from contract.

```
contract A {
    function deployB() public returns (address) {
        return address(new B());
    }
}

contract B {

}
```

Accessing public members of another contract is different from most languages. You should use the style of invocation.

```
contract A {
    function accessBMembers(B b) public {
        int c = b.x();
        bool d = b.y("param1", 2);
    }
}

contract B {
    int public x;
    mapping(string=>mapping(int=>bool)) public y;
}
```

`CALL` and `DELEGATECALL` are methods to call functions only with function names and the contract address. These are not recommended but can be useful.

Testing

Import `remix_tests.sol`, and `remix_accounts.sol` if you want to use accounts.

Make your test class be a subclass of the contract, and use assertions in each tests. You may refer to the documentation of Remix Assert Library.

```
pragma solidity >=0.7.0 <0.9.0;
import "remix_tests.sol";
import "remix_accounts.sol";
import "../contracts/3_Ballot.sol";

contract ExampleTest is Example {

    address acc0;
    address acc1;
    address acc2;
    address acc3;
    address acc4;

    function beforeAll() public {
        acc0 = TestsAccounts.getAccount(0);
        acc1 = TestsAccounts.getAccount(1);
        acc2 = TestsAccounts.getAccount(2);
        acc3 = TestsAccounts.getAccount(3);
        acc4 = TestsAccounts.getAccount(4);
    }

    /// #sender: account-3
    function test1() public {
        Assert.equal(f(), 5, "execution result of f should be 5");
    }
}
```

Additional information

The following features are not necessarily useful in project 1, but might be needed in project 2. You may want to explore these later.

- Interface and Inheritance: useful for writing reusable smart contracts
- Internal function: passing function-type parameters can be useful for writing a generic sort library
- String: strings operations in Solidity is still primitive, so you need to be cautious when using strings
- Type conversion: if the contract plays with integers heavily, you may find type conversion useful
- Library: libraries can share common operations of data types across smart contracts
- Payment: although not used in the project 1, the transfer of crytocurrency is an essential feature of blockchain
- Events: if you are building DApps containing active interactions between on-chain and off-chain components, you may make use of this important trigger; you may also emit an event to notify people the final result of lunch venue poll, but won't get extra marks for that ;)
- Internals and Inline assembly: learning internals is not required for this course, but could be extremely helpful for debugging and building advanced DApps

OpenZeppelin maintains a set of popular libraries for smart contract development on Ethereum. You may use these libraries in your project 2. You can also learn blockchain programming with their tutorials.

Last but not least, there is a contract code size limit, and you should consider separating the responsibilities of your contracts if you wrote a huge contract. You can follow the guide on Ethereum website.