Introduction to Solidity

Aggelos Kiayias

Dionysis Zindros, Christos Nasikas

Overview

- Types
- Data location
- Visibility
- Fallback function
- Send ether
- Interacting with other contracts

```
pragma solidity ^0.5.1;

contract HelloWorld {

 function print () public pure returns (string memory) {

    return 'Hello World!';

 }

}
```

Pragmas

```
pragma solidity 0.5.0;

pragma solidity ^0.5.1;

pragma solidity >=0.5.0 < 0.6.0;
```

The pragma keyword is used to enable certain compiler (version) features or checks. Follows the same syntax used by npm.

Contract

```
contract <contract-name> { … }
```

Constructors

```
contract HelloWorld {
    constructor () public { … }
}
```

```
contract HelloWorld {
    constructor (uint x, string y) public { … }
}
```

Types

- The **type** of each variable **needs to be specified** (Solidity is a statically typed language)
- **Two** types:
  - **Value** types
  - **Reference** types
- "**undefined**" or "**null**" values **does not exist** in Solidity
- **Variables** without a value **always** have a **default value** (zero-state) dependent on its type.
- Solidity follows the scoping rules of C99 (variables are visible until the end of the smallest `{}`-block)

Value types

Types: booleans

```
contract Booleans {
    bool p = true;
    bool q = false;
}
```

Operators: `!`, `&&`, `||`, `!=`, `==`

Types: integers

```
contract Integers {

    uint256 x = 5;

    int8 y = -5;

}
```

- Two types:
  - `int` (signed)
  - `uint` (unsigned)
- Keywords: `uint8` / `int8` to `uint256` / `int256` in step of 8.
- `uint` / `int` are alias for `uint256` / `int256`.
- Operators as usual:
  - Comparisons: `<=, <, ==, !=, >=, >`
  - Arithmetic operators: `+, -, *, /, %, **`
  - Bitwise operators: `&, |, ^`
  - Shift operators: `>>, <<`
- Range: $2^b - 1$ where $b \in \{ 8, 16, 24, 32, \ldots, 256 \}$
- Division always results in an integer and round towards zero (5 / 2 = 2).
- No floats!

Types: address

```
contract Address {
    address owner;
    address payable anotherAddress;
}
```

Address type holds an Ethereum address (20 byte value).
Payable address is an address you can send Ether to (you cannot send to plain addresses).

Types: fixed-size byte arrays

```
contract ByteArrays {

    bytes32 y = 0xa5b9...;

    // y.length == 32

}
```

- `bytes1`, `bytes2`, `bytes3`, ..., `bytes32`
- `byte` is alias for `byte1`
- `length`: fixed length of the byte array. You cannot change the length of a fixed byte array.

Types: Enum

```
contract Purchase {
    enum State { Created, Locked, Inactive }
}
```

Reference types

Types: arrays

```
contract Arrays {
     uint256[2] x;
     uint8[] y;
     bytes z;
     string name;
     // 2D: dynamic rows, 2 columns!
     uint [2][] flags;

     function create () public {
          uint[] memory a = new uint[](7);
          flags.push([0, 1]);
     }
}
```

- The **notation** of declaring **2D** arrays is **reversed** when compared to **other languages**!
    - **Declaration**: `uint[columns][rows] z;`
    - **Access**: `z[row][column]`
- `bytes` and `string` are **special** arrays.
- `bytes` is similar to `byte[]` but is **cheaper** (gas).
- `string` is a **UTF-8-encoded**.
- Members:
    - push: push an element at the end of array.
    - length: return or set the size of array.
- `string` does **not** have **length** member.
- **Allocate** memory **arrays** by using the **keyword** `new`. The size of memory arrays has to be known at compilation. You **cannot** resize a memory array.

Types: Struct

```
contract Vote {
    struct Voter {
        bool voted;
        address voter;
        uint vote;
    }
}
```

- A struct cannot contain a struct of its own type (the size of the struct has to be finite).
- A struct can contain mappings.

Types: Mappings

```
contract Mappings {

    mapping(address => uint256) balances;

}
```

key

value

Visibility

Visibility

- **public**: Public functions can be called from other contracts, internally and personal accounts. For public state variables an automatic getter function is being created.
- **external**: External functions cannot be called internally. Variables cannot be declared as external.
- **Internal**: Internal function and variables can be called only internally. Contracts that inherit another contract can access the parent's internal variables and functions.
- **private**: Private functions and variables can be called only by the contract in which they are defined and not in derived contract. **Warning:** private variables are visible to all observers external to the blockchain.

Data location

Data location: areas

- Every complex type (arrays, structs, mappings) have a data location.
- Two types of location: storage and memory.
- As of Solidity version **5.0.0** you must **always declare** the data **location** of complex types inside functions’ body, arguments and returned values.

Data location: areas

- Storage:
    - Persistent.
    - All state variables are saved to storage.
    - Function’s complex local values are saved to storage by default. (Solidity versions >= 5.0.0 force you to declare the data location).
- Memory:
    - Non-persistent.
    - Function’s arguments and returned values are stored to memory by default. (Solidity versions >= 5.0.0 force you to declare the data location for complex types).

Data location: assignment

- storage <-> memory: copy
- state variable <- state variable, storage and memory: copy
- memory <-> memory : reference
- local storage variable <- storage: reference

Fallback function

Fallback function

```
contract Fallback {

    function () external {

        ...

    }

}
```

Unnamed function

- No arguments (`msg.data` is accessible).
- No returned values.
- Mandatory visibility: external.
- Executed if no data (transaction field) is supplied or if the function that a user tries to call does not exist.
- Executed whenever the contract receives plain Ether (without data).
- To receive Ether the fallback function must be marked as `payable`.
- In the absence of fallback function a contract cannot receive Ether and an exception is thrown.

Send ether

Send ether

Function	Gas forwarded	Error handling	Notes
`transfer`	2300	throws on failure	**Safe** against re-entrancy
`send`	2300	`false` on failure	**Safe** against re-entrancy
`call`	all remaining gas	`false` on failure	**Not safe** against re-entrancy

Interacting with other contracts

Interacting with other contracts

```
contract Planet {
    string private name;
    constructor (string memory _name) public { name = _name; }
    function getName() public returns(string memory) { return name; }
}

contract Universe {
    address[] planets;
    event NewPlanet(address planet, string name);

    function createNewPlanet(string memory name) public {
        Planet p = new Planet(name);
        planets.push(address(p));
        emit NewPlanet(address(p), p.getName());
    }
}
```

Thank you!