Smart Contracts and Blockchain Technology

Lecture 11. Control structures in Solidity

Christian Ewerhart

University of Zurich

Fall 2022

Introduction and overview

Last lecture: Core language concepts

- Functions
- Contracts

This lecture: Advanced elements of Solidity

- Error handling
- Data types (cont.)

Control structures in Solidity (1)

Error handling

There are three ways to (conditionally) trigger an error:

- `assert`
- `require`
- `revert`

In addition, errors may be caught by calling contracts:

- `try` and `catch`

Control structures in Solidity (2)

Assert

To check a condition that should be true for any input, you may use the **assert function**:

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity >=0.8.0 <0.9.0;
contract c {
    function k(string memory s) public pure returns (bytes32) {
        return keccak256(abi.encodePacked(s));
    }

    // check if keccak256 works
    bytes32 constant emptyString = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
    function GetItRight() public pure {
        assert(emptyString == keccak256(abi.encodePacked("")));
    }
}
```

If the condition is not met, a panic error is triggered.

Control structures in Solidity (3)

Require

To check a condition, you may use the **require function**:

```
require(msg.sender == owner; "not authorized");
```

Control structures in Solidity (4)

Revert function

To trigger an unconditional error, you may use the **revert function**:[1]

```
revert("This should not have happened!");
```

The error data will be passed back to the caller and can be caught there using a try/catch statement.

[1] To save gas, one may alternatively define an error message and call the **revert statement**.

Control structures in Solidity (5)

Try/catch

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
interface DataFeed { function getData(address token) external returns (uint value); }
contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case revert was called inside getData and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic, i.e. a serious error like division by zero or overflow.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used.
            errorCount++;
            return (0, false);
        }
    }
}
```

Control structures in Solidity (6)

Data types (cont.)

Reference types:

- Structs
- Arrays and strings
- Mappings

If you use a reference type, you always have to explicitly provide the data area where the type is stored:

- **storage** (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract), or
- **memory** (whose lifetime is limited to an external function call),
- **calldata** (= stack, special data location that contains the function arguments).

Control structures in Solidity (7)

Storage and memory

Storage is like a computer hard drive. State variables are storage data. These state variables reside in the smart contract data section on the blockchain. Writing variables into storage is expensive (in terms of gas consumption).

Memory is a temporary place to store data, like RAM. Function arguments and local variables in functions are memory data. If the function is external, args will be stored in the stack (calldata). EVM has limited space for memory so values stored here are erased between function calls.

Control structures in Solidity (8)

Structs

New data types may be defined in the form of **structs**:

```
struct Book {
    string title;
    string author;
    uint book_id;
}
```

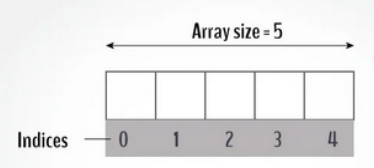
```
Book storage book; // variable of type Book
```

```
book.title = "The Tell-Tale heart";
book.author = "Edgar Allen Poe";
book.book_id = 1;
```

Control structures in Solidity (9)

Arrays

Storage arrays either have a **fixed size** or a **dynamic size**:[2]




```
uint[6] a; // declares a static array
uint[] d; // declares a dynamic array
bytes b; // arbitary-length byte data
string s; // arbitary-length UTF-8 data
a[4] = 8; // assigns a value to array element
```

[2] Memory arrays are static once created (but the size can be set at runtime).

Control structures in Solidity (10)

Unicode Transformation Format 8 bit (UTF-8)

UTF-8 is the predominant character encoding.

UTF-8 is backward compatible with ASCII but variable-length:

- The first 128 characters of Unicode correspond one-to-one with ASCII. E.g., “$” = 0x24.
- The checksum bit in ASCII is reinterpreted to signal that more bytes are used to encode the character. E.g., “€” = 0xe282ac.

Examples of UTF-8 encoding

Character		Binary code point	Binary UTF-8	Hex UTF-8
$	U+0024	010 0100	00100100	24
£	U+00A3	000 1010 0011	11000010 10100011	C2 A3
ह	U+0939	0000 1001 0011 1001	11100000 10100100 10111001	E0 A4 B9
€	U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	E2 82 AC
한	U+D55C	1101 0101 0101 1100	11101101 10010101 10011100	ED 95 9C
𐍈	U+10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000	F0 90 8D 88

Control structures in Solidity (11)

Fixed-size byte arrays

Raw byte code of given size may be represented by **fixed-size byte arrays**:

`bytes1, bytes2, ..., bytes32`

The number indicates the number of bytes.

Control structures in Solidity (12)

Strings

```
pragma solidity ^0.5.0;
contract SolidityTest {
    string data = "test";
}
```

Control structures in Solidity (13)

Methods for dynamic storage arrays

On dynamic storage arrays:

- `.length` returns the number of elements of the array.
- `.push(value)` appends `value` at the end of the array.
- `.push()` appends a zero-initialised element.
- `.pop()` removes the element at the end of the array.

Note: `.push(value)`, `.push()`, and `.pop()` do not work for strings.

Control structures in Solidity (14)

Mappings

Mappings allow to create and manage lists in a flexible way.

Example:

```
contract LedgerBalance {
   mapping(address => uint) public balances;
   function updateBalance(uint newBalance) public {
     balances[msg.sender] = newBalance;
   }
}
```

Control structures in Solidity (15)

Mappings (continued)

Variables of **mapping type** are declared using the syntax

```
mapping(KeyType => ValueType) VariableName
```

The `KeyType` can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed.

`ValueType` can be any type, including mappings, arrays and structs.

Note: Mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information.

Control structures in Solidity (16)

Enum types

New data types may be defined in the form of **enums**:

```
contract softdrinks {
    enum Size{ SMALL, MEDIUM, LARGE }
    Size choice; // variable of type Size
    function setChoice() public {
        choice = Size.LARGE;
    }
    function getChoice() public returns (Size) {
        return choice;
    }
}
```

Control structures in Solidity (17)

State variable visibility

`private`
...can only be accessed from within the contract they are declared in.

`internal`
...can be accessed both from within the contract they are declared in and in derived contracts.[3] This is the default visibility level for state variables.

`public`
...differ from internal ones in that the compiler automatically generates **getter functions** for them.[4]

---

[3] Derived contracts inherit from existing **base contracts**. They are defined using the keyword `is` in the contract declaration.

[4] For instance, to read the value of variable `x` in a contract `myContract`, one uses `myContract.x`.

Outlook on the remaining lectures

Lecture 12: Token programming

- ERC20 tokens
- Alternative token standards

Lecture 13: Decentralized finance

- Decentralized exchanges
- Mixers

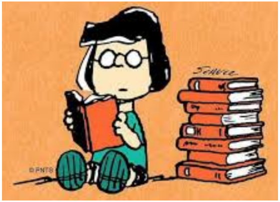
Lecture 14: Beyond crypto

- Use cases
- Future of blockchain technology

Control structures in Solidity (18)

Bibliographic notes

This slide deck is based on Section 3.9 of the official documentation of the programming language Solidity (link).



Control structures in Solidity (19)

References

Solidity Documentation, **Release 0.8.17**, Ethereum Foundation, September 8, 2022.