Smart Contracts and Blockchain Technology

Lecture 9. Solidity variables and types

Christian Ewerhart

University of Zurich

Fall 2022

Introduction and overview

Last lecture: Smart contracts

This lecture: Solidity variables and types

- Value types
- Reference types
- Visibility

Solidity variables and types (1)

Type declarations

Solidity is a **statically typed programming language,** which means that the type of each variable needs to be specified in the form of an **explicit type declaration** at compile time.[1]

Moreover, newly declared variables always have a **default value:**

```
uint x; // declares x and sets it to 0x0
```

[1] In contrast, *Python* is a dynamically typed language because it does not require the declaration of variables at compile time, and the checking of types takes place at runtime.

Solidity variables and types (2)

Value types vs. reference types

Solidity distinguishes between value types and reference types:

- Variables of **value types** are always passed on by value, i.e., they are copied when they are used as function arguments or in assignments.
- Variables of **reference type** can be addressed through multiple, different names.[2]

[2] It is useful to think of variables of the reference types as *pointers*, so that several pointers may identify the same data.

Solidity variables and types (3)

Value types

- Boolean
- Integer
- Fixed-point number[3]
- Address (payable)
- Contract
- Fixed-size byte array
- Literal
- Enum
- User defined
- Function

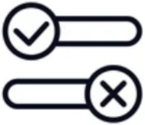
[3] Fixed-point numbers are partially supported only. Floating-point numbers (i.e., scientific notation) are not supported at all.

Solidity variables and types (4)

Booleans

Booleans represent precisely one bit of information. Correspondingly, they take one of two values, `true` or `false`.

```
bool flag; // declares a boolean
```



Note: The value of any boolean is initially `false`.

```
flag = true; // flag is set
```

Solidity variables and types (5)

Use of booleans in a control structure

Booleans may be used in control structures, for instance:[4]

```
if flag == true {
    // do something
}
```

Equivalent is the following:

```
if flag {
    // do something
}
```

[4] Control structures are discussed in the next lecture.

Solidity variables and types (6)

Some operators

! “not” (logical negation)

&& “and” (logical conjunction)

|| “or” (logical disjunction)

!= “not equal to” (inequality)

Solidity variables and types (7)

Integers

Unsigned integers

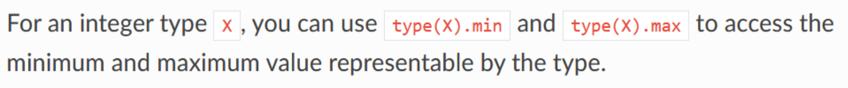
`uint`

`uint8, uint16, uint24, ..., uint256`[5]

Signed integers

`int`

`int8, int16, int24, ..., int256`[6]



For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

[5] The number indicates the number of bits in the machine-level representation. The type `uint256` is equivalent to `uint`.

[6] The type `int256` is equivalent to `int`.

Solidity variables and types (8)

Type conversions

An **implicit conversion** is applied by the compiler if it makes sense semantically and no information is lost:

- during assignments,
- when applying operators,
- when passing arguments to functions.

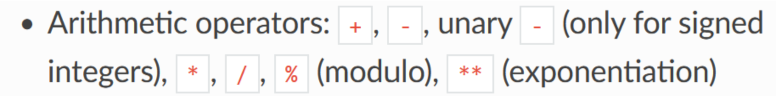
```
uint8 x;
uint16 y;
uint32 z = x + y; // this works
```

An **explicit type conversion** adds clarity at the cost of formalism:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
```

Solidity variables and types (9)

Integer operations



- Arithmetic operators: `+`, `-`, unary `-` (only for signed integers), `*`, `/`, `%` (modulo), `**` (exponentiation)

Division or modulo by zero causes a so-called panic error.[7]

[7] As will be discussed, an error will undo all changes made to the state during a message call, and it will "bubble up" unless caught by a `try` instruction.

Solidity variables and types (10)

Address

An Ethereum address is a 20 byte hexadecimal value.

```
address constant customer1 =
0xc0ffee254729296a45a3885639AC7E10F9d54979;
```

It is possible to query the ether balance of an address using the property `.balance`.

```
if customer1.balance == 0 revert();
```

Solidity variables and types (11)

Members of address

`.balance` returns the **balance** of the address in Wei as `uint256`.

`.code` returns the **code** at the address as `bytes memory` (can be empty).[8]

`.codehash` returns the Keccak-256 hash of the code at address as `bytes32`.[9]

[8] This allows to check if an address is a contract address or not.

[9] This can be used to check if the code on a given address is as expected.

Solidity variables and types (12)

Low-level contract calls

Any call method:

- operates on an address (rather than a contract instance),
- takes a given `payload` as `bytes memory` argument,
- returns success condition and return data as `(bool, bytes memory)`, and
- forwards all available gas, adjustable.

```
.call(payload)
```

```
.delegatecall(payload)
```

`.staticcall(payload)`[10]

[10] This is basically the same as `call`, but will revert if the called function modifies the state in any way.

Solidity variables and types (13)

Address payable

A contract can make a payment only to an `address` that has been declared `payable`.[11]

```
address payable borrower;
borrower = payable(customer1); // conversion needed
```

`address payable` has two additional members, both sending an `amount` in Wei as `uint256` and forwarding a 2300 gas stipend (not adjustable):

`.transfer(amount)` reverts on failure (either payment not accepted or insufficient funds).

`.send(amount)` returns a `bool` (equal to `false` on failure).

[11] Implicit conversion from payable to address is possible, but not vice versa.

Solidity variables and types (14)

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;
    }
}
```

Solidity variables and types (15)

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.

Solidity variables and types (16)

Reference types

Reference types:

- Structs
- Arrays
- 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** (special data location that contains the function arguments).

Solidity variables and types (17)

Structs

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

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

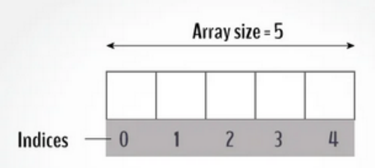
```
Book storage book; // variable of type Book
```

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

Solidity variables and types (18)

Arrays

Storage arrays either have a **fixed size** or a **dynamic size**:[12]



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

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

Solidity variables and types (19)

Strings

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

Solidity variables and types (20)

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.

Solidity variables and types (21)

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;
   }
}
```

Solidity variables and types (22)

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.

Solidity variables and types (23)

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.[13] This is the default visibility level for state variables.

`public`

...differ from internal ones in that the compiler automatically generates **getter functions** for them, which allows other contracts to read their values.[14]

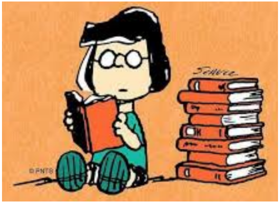
[13] Derived contracts will be discussed in the next lecture.

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

Solidity variables and types (24)

Bibliographic notes

This slide deck is based on Section 3.6 of the official documentation of the programming language Solidity (link).



Solidity variables, types, and expressions (25)

References

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