**Getter functions**

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called data that does not take any arguments and returns a uint, the value of the state variable data. State variables can be initialized when they are declared.

*// SPDX-License-Identifier: GPL-3.0*

**pragma solidity** >=**0.4.16** <**0.9.0**;

**contract** **C** {

uint **public** data = 42;

}

**contract** **Caller** {

C c = new C();

function f() public view returns (uint) {

return c.data();

}

}

The getter functions have external visibility. If the symbol is accessed internally (i.e. without this.), it evaluates to a state variable. If it is accessed externally (i.e. **with this.), it evaluates to a function.**

*// SPDX-License-Identifier: GPL-3.0*

**pragma solidity** >=**0.4.0** <**0.9.0**;

**contract** **C** {

uint **public** data;

function x() public returns (uint) {

data = 3; *// internal access*

return this.data(); *// external access*

}

}

**Struct**

Cannot create constructor for struct.

Struct can be defined outside contract so that it can be accessible in entire file.

*// SPDX-License-Identifier: GPL-3.0*

**pragma solidity** >=**0.4.0** <**0.9.0**;

**contract** **Complex** {

struct Data {

uint a;

bytes3 b;

mapping (uint => uint) map;

uint[3] c;

uint[] d;

bytes e;

}

mapping (uint => mapping(bool => Data[])) public data;

}

For public state variables, getters are automatically created.

It generates a function internally of the following form.

function data(uint arg1, bool arg2, uint arg3)

public

returns (uint a, bytes3 b, bytes memory e)

{

a = data[arg1][arg2][arg3].a;

b = data[arg1][arg2][arg3].b;

e = data[arg1][arg2][arg3].e;

}

The mapping and arrays (with the exception of byte arrays) inside the struct are omitted when getter functions are created because there is no good way to select individual struct members or provide a key for the mapping:

Initialization example:

struct fooStruct {

uint foo;

uint figther;

}

Yes, just use

fooStruct myStruct = fooStruct(1,2);

Or

fooStruct myStruct = fooStruct({foo:1, fighter:2});

Or

fooStruct memory myStruct; // for temporary data

myStruct.figther = 2; // will only write to memory

fooStruct storage myStruct = ...; // for persistent data, has to be initialized from a state variable. `storage` is the default and a warning will be thrown by Solidity compiler versions starting with 4.17

myStruct.fighter = 2; // will write directly to storage