**Constant and Immutable State Variables**[**ℑ**](https://docs.soliditylang.org/en/v0.8.15/contracts.html#constant-and-immutable-state-variables)

State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time.

It is also possible to define constant variables at the file level.

**The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value**.

**Compared to regular state variables, the gas costs of constant and immutable variables are much lower. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations.**

**Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.**

Not all types for constants and immutables are implemented at this time. The only supported types are [strings](https://docs.soliditylang.org/en/v0.8.15/types.html#strings) (only for constants) and [value types](https://docs.soliditylang.org/en/v0.8.15/types.html#value-types).

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

**pragma solidity** >=**0.7.4**;

uint constant X = 32\*\*22 + 8;

**contract** **C** {

string constant TEXT = "abc";

bytes32 constant MY\_HASH = keccak256("abc");

uint immutable decimals;

uint immutable maxBalance;

address immutable owner = **msg.sender**;

constructor(uint decimals\_, address ref) {

decimals = decimals\_;

*// Assignments to immutables can even access the environment.*

maxBalance = ref.balance;

}

function isBalanceTooHigh(address other) public view returns (bool) {

return other.balance > maxBalance;

}

}

**Constant**[**ℑ**](https://docs.soliditylang.org/en/v0.8.15/contracts.html#constant)

For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. block.timestamp, address(this).balance or block.number) or execution data (msg.value or gasleft()) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not.

The built-in functions keccak256, sha256, ripemd160, ecrecover, addmod and mulmod are allowed (even though, with the exception of keccak256, they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

**Immutable**[**ℑ**](https://docs.soliditylang.org/en/v0.8.15/contracts.html#immutable)

Variables declared as immutable are a bit less restricted than those declared as constant: **Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They can be assigned only once** and can, from that point on, be read even during construction time.

The contract creation code generated by the compiler will modify the contract’s runtime code before it is returned by replacing all references to immutables by the values assigned to the them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

**Note**

Immutables that are assigned at their declaration are only considered initialized once the constructor of the contract is executing. This means you cannot initialize immutables inline with a value that depends on another immutable. You can do this, however, inside the constructor of the contract.

This is a safeguard against different interpretations about the order of state variable initialization and constructor execution, especially with regards to inheritance.