**Solidity IR-based Codegen Changes**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#solidity-ir-based-codegen-changes)

Solidity can generate EVM bytecode in two different ways: Either directly from Solidity to EVM opcodes (“old codegen”) or through an intermediate representation (“IR”) in Yul (“new codegen” or “IR-based codegen”).

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.

You can enable it on the command line using --via-ir or with the option {"viaIR": true} in standard-json and we encourage everyone to try it out!

For several reasons, there are tiny semantic differences between the old and the IR-based code generator, mostly in areas where we would not expect people to rely on this behaviour anyway. This section highlights the main differences between the old and the IR-based codegen.

**Semantic Only Changes**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#semantic-only-changes)

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

* The order of state variable initialization has changed in case of inheritance.

The order used to be:

* + All state variables are zero-initialized at the beginning.
  + Evaluate base constructor arguments from most derived to most base contract.
  + Initialize all state variables in the whole inheritance hierarchy from most base to most derived.
  + Run the constructor, if present, for all contracts in the linearized hierarchy from most base to most derived.

New order:

* + All state variables are zero-initialized at the beginning.
  + Evaluate base constructor arguments from most derived to most base contract.
  + For every contract in order from most base to most derived in the linearized hierarchy:
    1. Initialize state variables.
    2. Run the constructor (if present).

This causes differences in contracts where the initial value of a state variable relies on the result of the constructor in another contract:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC43LjE7Cgpjb250cmFjdCBBIHsKICAgIHVpbnQgeDsKICAgIGNvbnN0cnVjdG9yKCkgewogICAgICAgIHggPSA0MjsKICAgIH0KICAgIGZ1bmN0aW9uIGYoKSBwdWJsaWMgdmlldyByZXR1cm5zKHVpbnQyNTYpIHsKICAgICAgICByZXR1cm4geDsKICAgIH0KfQpjb250cmFjdCBCIGlzIEEgewogICAgdWludCBwdWJsaWMgeSA9IGYoKTsKfQ==)

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

**pragma solidity** >=**0.7.1**;

**contract** **A** {

uint x;

constructor() {

x = 42;

}

function f() public view returns(uint256) {

return x;

}

}

**contract** **B** is A {

uint **public** y = f();

}

Previously, y would be set to 0. This is due to the fact that we would first initialize state variables: First, x is set to 0, and when initializing y, f() would return 0 causing y to be 0 as well. With the new rules, y will be set to 42. We first initialize x to 0, then call A’s constructor which sets x to 42. Finally, when initializing y, f() returns 42 causing y to be 42.

* When storage structs are deleted, every storage slot that contains a member of the struct is set to zero entirely. Formerly, padding space was left untouched. Consequently, if the padding space within a struct is used to store data (e.g. in the context of a contract upgrade), you have to be aware that delete will now also clear the added member (while it wouldn’t have been cleared in the past).

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC43LjE7Cgpjb250cmFjdCBDIHsKICAgIHN0cnVjdCBTIHsKICAgICAgICB1aW50NjQgeTsKICAgICAgICB1aW50NjQgejsKICAgIH0KICAgIFMgczsKICAgIGZ1bmN0aW9uIGYoKSBwdWJsaWMgewogICAgICAgIC8vIC4uLgogICAgICAgIGRlbGV0ZSBzOwogICAgICAgIC8vIHMgb2NjdXBpZXMgb25seSBmaXJzdCAxNiBieXRlcyBvZiB0aGUgMzIgYnl0ZXMgc2xvdAogICAgICAgIC8vIGRlbGV0ZSB3aWxsIHdyaXRlIHplcm8gdG8gdGhlIGZ1bGwgc2xvdAogICAgfQp9)

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

**pragma solidity** >=**0.7.1**;

**contract** **C** {

struct S {

uint64 y;

uint64 z;

}

S s;

function f() public {

*// ...*

delete s;

*// s occupies only first 16 bytes of the 32 bytes slot*

*// delete will write zero to the full slot*

}

}

We have the same behavior for implicit delete, for example when array of structs is shortened.

* Function modifiers are implemented in a slightly different way regarding function parameters and return variables. This especially has an effect if the placeholder \_; is evaluated multiple times in a modifier. In the old code generator, each function parameter and return variable has a fixed slot on the stack. If the function is run multiple times because \_; is used multiple times or used in a loop, then a change to the function parameter’s or return variable’s value is visible in the next execution of the function. The new code generator implements modifiers using actual functions and passes function parameters on. This means that multiple evaluations of a function’s body will get the same values for the parameters, and the effect on return variables is that they are reset to their default (zero) value for each execution.

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC43LjA7CmNvbnRyYWN0IEMgewogICAgZnVuY3Rpb24gZih1aW50IGEpIHB1YmxpYyBwdXJlIG1vZCgpIHJldHVybnMgKHVpbnQgcikgewogICAgICAgIHIgPSBhKys7CiAgICB9CiAgICBtb2RpZmllciBtb2QoKSB7IF87IF87IH0KfQ==)

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

**pragma solidity** >=**0.7.0**;

**contract** **C** {

function f(uint a) public pure mod() returns (uint r) {

r = a++;

}

modifier mod() { \_; \_; }

}

If you execute f(0) in the old code generator, it will return 1, while it will return 0 when using the new code generator.

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC43LjEgPDAuOS4wOwoKY29udHJhY3QgQyB7CiAgICBib29sIGFjdGl2ZSA9IHRydWU7CiAgICBtb2RpZmllciBtb2QoKQogICAgewogICAgICAgIF87CiAgICAgICAgYWN0aXZlID0gZmFsc2U7CiAgICAgICAgXzsKICAgIH0KICAgIGZ1bmN0aW9uIGZvbygpIGV4dGVybmFsIG1vZCgpIHJldHVybnMgKHVpbnQgcmV0KQogICAgewogICAgICAgIGlmIChhY3RpdmUpCiAgICAgICAgICAgIHJldCA9IDE7IC8vIFNhbWUgYXMgYGByZXR1cm4gMWBgCiAgICB9Cn0=)

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

**pragma solidity** >=**0.7.1** <**0.9.0**;

**contract** **C** {

bool active = true;

modifier mod()

{

\_;

active = false;

\_;

}

function foo() external mod() returns (uint ret)

{

if (active)

ret = 1; *// Same as ``return 1``*

}

}

The function C.foo() returns the following values:

* + Old code generator: 1 as the return variable is initialized to 0 only once before the first \_; evaluation and then overwritten by the return 1;. It is not initialized again for the second \_; evaluation and foo() does not explicitly assign it either (due to active == false), thus it keeps its first value.
  + New code generator: 0 as all parameters, including return parameters, will be re-initialized before each \_; evaluation.
* For the old code generator, the evaluation order of expressions is unspecified. For the new code generator, we try to evaluate in source order (left to right), but do not guarantee it. This can lead to semantic differences.

For example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC44LjE7CmNvbnRyYWN0IEMgewogICAgZnVuY3Rpb24gcHJlaW5jcl91OCh1aW50OCBhKSBwdWJsaWMgcHVyZSByZXR1cm5zICh1aW50OCkgewogICAgICAgIHJldHVybiArK2EgKyBhOwogICAgfQp9)

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

**pragma solidity** >=**0.8.1**;

**contract** **C** {

function preincr\_u8(uint8 a) public pure returns (uint8) {

return ++a + a;

}

}

The function preincr\_u8(1) returns the following values:

* + Old code generator: 3 (1 + 2) but the return value is unspecified in general
  + New code generator: 4 (2 + 2) but the return value is not guaranteed

On the other hand, function argument expressions are evaluated in the same order by both code generators with the exception of the global functions addmod and mulmod. For example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC44LjE7CmNvbnRyYWN0IEMgewogICAgZnVuY3Rpb24gYWRkKHVpbnQ4IGEsIHVpbnQ4IGIpIHB1YmxpYyBwdXJlIHJldHVybnMgKHVpbnQ4KSB7CiAgICAgICAgcmV0dXJuIGEgKyBiOwogICAgfQogICAgZnVuY3Rpb24gZyh1aW50OCBhLCB1aW50OCBiKSBwdWJsaWMgcHVyZSByZXR1cm5zICh1aW50OCkgewogICAgICAgIHJldHVybiBhZGQoKythICsgKytiLCBhICsgYik7CiAgICB9Cn0=)

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

**pragma solidity** >=**0.8.1**;

**contract** **C** {

function add(uint8 a, uint8 b) public pure returns (uint8) {

return a + b;

}

function g(uint8 a, uint8 b) public pure returns (uint8) {

return add(++a + ++b, a + b);

}

}

The function g(1, 2) returns the following values:

* + Old code generator: 10 (add(2 + 3, 2 + 3)) but the return value is unspecified in general
  + New code generator: 10 but the return value is not guaranteed

The arguments to the global functions addmod and mulmod are evaluated right-to-left by the old code generator and left-to-right by the new code generator. For example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC44LjE7CmNvbnRyYWN0IEMgewogICAgZnVuY3Rpb24gZigpIHB1YmxpYyBwdXJlIHJldHVybnMgKHVpbnQyNTYgYU1vZCwgdWludDI1NiBtTW9kKSB7CiAgICAgICAgdWludDI1NiB4ID0gMzsKICAgICAgICAvLyBPbGQgY29kZSBnZW46IGFkZC9tdWxtb2QoNSwgNCwgMykKICAgICAgICAvLyBOZXcgY29kZSBnZW46IGFkZC9tdWxtb2QoNCwgNSwgNSkKICAgICAgICBhTW9kID0gYWRkbW9kKCsreCwgKyt4LCB4KTsKICAgICAgICBtTW9kID0gbXVsbW9kKCsreCwgKyt4LCB4KTsKICAgIH0KfQ==)

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

**pragma solidity** >=**0.8.1**;

**contract** **C** {

function f() public pure returns (uint256 aMod, uint256 mMod) {

uint256 x = 3;

*// Old code gen: add/mulmod(5, 4, 3)*

*// New code gen: add/mulmod(4, 5, 5)*

aMod = addmod(++x, ++x, x);

mMod = mulmod(++x, ++x, x);

}

}

The function f() returns the following values:

* + Old code generator: aMod = 0 and mMod = 2
  + New code generator: aMod = 4 and mMod = 0
* The new code generator imposes a hard limit of type(uint64).max (0xffffffffffffffff) for the free memory pointer. Allocations that would increase its value beyond this limit revert. The old code generator does not have this limit.

For example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=)

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

**pragma solidity** >0.8.0;

**contract** **C** {

function f() public {

uint[] memory arr;

*// allocation size: 576460752303423481*

*// assumes freeMemPtr points to 0x80 initially*

uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 - 31) / 32;

*// freeMemPtr overflows UINT64\_MAX*

arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);

}

}

The function f() behaves as follows:

* + Old code generator: runs out of gas while zeroing the array contents after the large memory allocation
  + New code generator: reverts due to free memory pointer overflow (does not run out of gas)

**Internals**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#internals)

**Internal function pointers**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#internal-function-pointers)

The old code generator uses code offsets or tags for values of internal function pointers. This is especially complicated since these offsets are different at construction time and after deployment and the values can cross this border via storage. Because of that, both offsets are encoded at construction time into the same value (into different bytes).

In the new code generator, function pointers use internal IDs that are allocated in sequence. Since calls via jumps are not possible, calls through function pointers always have to use an internal dispatch function that uses the switch statement to select the right function.

The ID 0 is reserved for uninitialized function pointers which then cause a panic in the dispatch function when called.

In the old code generator, internal function pointers are initialized with a special function that always causes a panic. This causes a storage write at construction time for internal function pointers in storage.

**Cleanup**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#cleanup)

The old code generator only performs cleanup before an operation whose result could be affected by the values of the dirty bits. The new code generator performs cleanup after any operation that can result in dirty bits. The hope is that the optimizer will be powerful enough to eliminate redundant cleanup operations.

For example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC44LjE7CmNvbnRyYWN0IEMgewogICAgZnVuY3Rpb24gZih1aW50OCBhKSBwdWJsaWMgcHVyZSByZXR1cm5zICh1aW50IHIxLCB1aW50IHIyKQogICAgewogICAgICAgIGEgPSB+YTsKICAgICAgICBhc3NlbWJseSB7CiAgICAgICAgICAgIHIxIDo9IGEKICAgICAgICB9CiAgICAgICAgcjIgPSBhOwogICAgfQp9)

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

**pragma solidity** >=**0.8.1**;

**contract** **C** {

function f(uint8 a) public pure returns (uint r1, uint r2)

{

a = ~a;

assembly {

r1 := a

}

r2 = a;

}

}

The function f(1) returns the following values:

* Old code generator: (fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe, 00000000000000000000000000000000000000000000000000000000000000fe)
* New code generator: (00000000000000000000000000000000000000000000000000000000000000fe, 00000000000000000000000000000000000000000000000000000000000000fe)

Note that, unlike the new code generator, the old code generator does not perform a cleanup after the bit-not assignment (a = ~a). This results in different values being assigned (within the inline assembly block) to return value r1 between the old and new code generators. However, both code generators perform a cleanup before the new value of a is assigned to r2.