**Inline Assembly**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#inline-assembly)

You can interleave Solidity statements with inline assembly in a language close to the one of the Ethereum Virtual Machine. This gives you more fine-grained control, which is especially useful when you are enhancing the language by writing libraries.

The language used for inline assembly in Solidity is called [Yul](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul) and it is documented in its own section. This section will only cover how the inline assembly code can interface with the surrounding Solidity code.

**Warning**

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

An inline assembly block is marked by assembly { ... }, where the code inside the curly braces is code in the [Yul](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul) language.

The inline assembly code can access local Solidity variables as explained below.

Different inline assembly blocks share no namespace, i.e. it is not possible to call a Yul function or access a Yul variable defined in a different inline assembly block.

**Example**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#example)

The following example provides library code to access the code of another contract and load it into a bytes variable. This is possible with “plain Solidity” too, by using <address>.code. But the point here is that reusable assembly libraries can enhance the Solidity language without a compiler change.

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

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

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

library GetCode {

function at(address addr) public view returns (bytes memory code) {

assembly {

*// retrieve the size of the code, this needs assembly*

let size := extcodesize(addr)

*// allocate output byte array - this could also be done without assembly*

*// by using code = new bytes(size)*

code := mload(0x40)

*// new "memory end" including padding*

mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))

*// store length in memory*

mstore(code, size)

*// actually retrieve the code, this needs assembly*

extcodecopy(addr, add(code, 0x20), 0, size)

}

}

}

Inline assembly is also beneficial in cases where the optimizer fails to produce efficient code, for example:

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

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

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

library VectorSum {

*// This function is less efficient because the optimizer currently fails to*

*// remove the bounds checks in array access.*

function sumSolidity(uint[] memory data) public pure returns (uint sum) {

for (uint i = 0; i < data.length; ++i)

sum += data[i];

}

*// We know that we only access the array in bounds, so we can avoid the check.*

*// 0x20 needs to be added to an array because the first slot contains the*

*// array length.*

function sumAsm(uint[] memory data) public pure returns (uint sum) {

for (uint i = 0; i < data.length; ++i) {

assembly {

sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))

}

}

}

*// Same as above, but accomplish the entire code within inline assembly.*

function sumPureAsm(uint[] memory data) public pure returns (uint sum) {

assembly {

*// Load the length (first 32 bytes)*

let len := mload(data)

*// Skip over the length field.*

*//*

*// Keep temporary variable so it can be incremented in place.*

*//*

*// NOTE: incrementing data would result in an unusable*

*// data variable after this assembly block*

let dataElementLocation := add(data, 0x20)

*// Iterate until the bound is not met.*

for

{ let end := add(dataElementLocation, mul(len, 0x20)) }

lt(dataElementLocation, end)

{ dataElementLocation := add(dataElementLocation, 0x20) }

{

sum := add(sum, mload(dataElementLocation))

}

}

}

}

**Access to External Variables, Functions and Libraries**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#access-to-external-variables-functions-and-libraries)

You can access Solidity variables and other identifiers by using their name.

Local variables of value type are directly usable in inline assembly. They can both be read and assigned to.

Local variables that refer to memory evaluate to the address of the variable in memory, not the value itself. Such variables can also be assigned to, but note that an assignment will only change the pointer and not the data and that it is your responsibility to respect Solidity’s memory management. See [Conventions in Solidity](https://docs.soliditylang.org/en/v0.8.20/assembly.html#conventions-in-solidity).

Similarly, local variables that refer to statically-sized calldata arrays or calldata structs evaluate to the address of the variable in calldata, not the value itself. The variable can also be assigned a new offset, but note that no validation is performed to ensure that the variable will not point beyond calldatasize().

For external function pointers the address and the function selector can be accessed using x.address and x.selector. The selector consists of four right-aligned bytes. Both values can be assigned to. For example:

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

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

**pragma solidity** >=**0.8.10** <**0.9.0**;

**contract** **C** {

*// Assigns a new selector and address to the return variable @fun*

function combineToFunctionPointer(address newAddress, uint newSelector) public pure returns (function() external fun) {

assembly {

fun.selector := newSelector

fun.address := newAddress

}

}

}

For dynamic calldata arrays, you can access their calldata offset (in bytes) and length (number of elements) using x.offset and x.length. Both expressions can also be assigned to, but as for the static case, no validation will be performed to ensure that the resulting data area is within the bounds of calldatasize().

For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their “address” is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable x, you use x.slot, and to retrieve the byte-offset you use x.offset. Using x itself will result in an error.

You can also assign to the .slot part of a local storage variable pointer. For these (structs, arrays or mappings), the .offset part is always zero. It is not possible to assign to the .slot or .offset part of a state variable, though.

Local Solidity variables are available for assignments, for example:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IEdQTC0zLjAKcHJhZ21hIHNvbGlkaXR5ID49MC43LjAgPDAuOS4wOwoKY29udHJhY3QgQyB7CiAgICB1aW50IGI7CiAgICBmdW5jdGlvbiBmKHVpbnQgeCkgcHVibGljIHZpZXcgcmV0dXJucyAodWludCByKSB7CiAgICAgICAgYXNzZW1ibHkgewogICAgICAgICAgICAvLyBXZSBpZ25vcmUgdGhlIHN0b3JhZ2Ugc2xvdCBvZmZzZXQsIHdlIGtub3cgaXQgaXMgemVybwogICAgICAgICAgICAvLyBpbiB0aGlzIHNwZWNpYWwgY2FzZS4KICAgICAgICAgICAgciA6PSBtdWwoeCwgc2xvYWQoYi5zbG90KSkKICAgICAgICB9CiAgICB9Cn0=)

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

**pragma solidity** >=**0.7.0** <**0.9.0**;

**contract** **C** {

uint b;

function f(uint x) public view returns (uint r) {

assembly {

*// We ignore the storage slot offset, we know it is zero*

*// in this special case.*

r := mul(x, sload(b.slot))

}

}

}

**Warning**

If you access variables of a type that spans less than 256 bits (for example uint64, address, or bytes16), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important: uint32 x = f(); assembly { x := and(x, 0xffffffff) /\* now use x \*/ } To clean signed types, you can use the signextend opcode: assembly { signextend(<num\_bytes\_of\_x\_minus\_one>, x) }

Since Solidity 0.6.0, the name of a inline assembly variable may not shadow any declaration visible in the scope of the inline assembly block (including variable, contract and function declarations).

Since Solidity 0.7.0, variables and functions declared inside the inline assembly block may not contain ., but using . is valid to access Solidity variables from outside the inline assembly block.

**Things to Avoid**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#things-to-avoid)

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

**Conventions in Solidity**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#conventions-in-solidity)

**Values of Typed Variables**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#values-of-typed-variables)

In contrast to EVM assembly, Solidity has types which are narrower than 256 bits, e.g. uint24. For efficiency, most arithmetic operations ignore the fact that types can be shorter than 256 bits, and the higher-order bits are cleaned when necessary, i.e., shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher-order bits first.

**Memory Management**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#memory-management)

Solidity manages memory in the following way. There is a “free memory pointer” at position 0x40 in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet you can use for allocating memory that follows the process outlined above:

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ZnVuY3Rpb24gYWxsb2NhdGUobGVuZ3RoKSAtPiBwb3MgewogIHBvcyA6PSBtbG9hZCgweDQwKQogIG1zdG9yZSgweDQwLCBhZGQocG9zLCBsZW5ndGgpKQp9)

**function** allocate(length) -> pos {

pos := mload(0x40)

mstore(0x40, add(pos, length))

}

The first 64 bytes of memory can be used as “scratch space” for short-term allocation. The 32 bytes after the free memory pointer (i.e., starting at 0x60) are meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at 0x80, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for bytes1[], but not for bytes and string). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

**Warning**

Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically and dynamically-sized arrays; so, do not rely on this.

**Memory Safety**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/assembly.html#memory-safety)

Without the use of inline assembly, the compiler can rely on memory to remain in a well-defined state at all times. This is especially relevant for [the new code generation pipeline via Yul IR](https://docs.soliditylang.org/en/v0.8.20/ir-breaking-changes.html#ir-breaking-changes): this code generation path can move local variables from stack to memory to avoid stack-too-deep errors and perform additional memory optimizations, if it can rely on certain assumptions about memory use.

While we recommend to always respect Solidity’s memory model, inline assembly allows you to use memory in an incompatible way. Therefore, moving stack variables to memory and additional memory optimizations are, by default, globally disabled in the presence of any inline assembly block that contains a memory operation or assigns to Solidity variables in memory.

However, you can specifically annotate an assembly block to indicate that it in fact respects Solidity’s memory model as follows:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=YXNzZW1ibHkgKCJtZW1vcnktc2FmZSIpIHsKICAgIC4uLgp9)

assembly ("memory-safe") {

...

}

In particular, a memory-safe assembly block may only access the following memory ranges:

* Memory allocated by yourself using a mechanism like the allocate function described above.
* Memory allocated by Solidity, e.g. memory within the bounds of a memory array you reference.
* The scratch space between memory offset 0 and 64 mentioned above.
* Temporary memory that is located *after* the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is “allocated” at the free memory pointer without updating the free memory pointer.

Furthermore, if the assembly block assigns to Solidity variables in memory, you need to assure that accesses to the Solidity variables only access these memory ranges.

Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates. As an example, the following assembly snippet is not memory safe, because the value of returndatasize() may exceed the 64 byte scratch space:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=YXNzZW1ibHkgewogIHJldHVybmRhdGFjb3B5KDAsIDAsIHJldHVybmRhdGFzaXplKCkpCiAgcmV2ZXJ0KDAsIHJldHVybmRhdGFzaXplKCkpCn0=)

assembly {

returndatacopy(0, 0, returndatasize())

revert(0, returndatasize())

}

On the other hand, the following code *is* memory safe, because memory beyond the location pointed to by the free memory pointer can safely be used as temporary scratch space:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=YXNzZW1ibHkgKCJtZW1vcnktc2FmZSIpIHsKICBsZXQgcCA6PSBtbG9hZCgweDQwKQogIHJldHVybmRhdGFjb3B5KHAsIDAsIHJldHVybmRhdGFzaXplKCkpCiAgcmV2ZXJ0KHAsIHJldHVybmRhdGFzaXplKCkpCn0=)

assembly ("memory-safe") {

let p := mload(0x40)

returndatacopy(p, 0, returndatasize())

revert(p, returndatasize())

}

Note that you do not need to update the free memory pointer if there is no following allocation, but you can only use memory starting from the current offset given by the free memory pointer.

If the memory operations use a length of zero, it is also fine to just use any offset (not only if it falls into the scratch space):

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=YXNzZW1ibHkgKCJtZW1vcnktc2FmZSIpIHsKICByZXZlcnQoMCwgMCkKfQ==)

assembly ("memory-safe") {

revert(0, 0)

}

Note that not only memory operations in inline assembly itself can be memory-unsafe, but also assignments to Solidity variables of reference type in memory. For example the following is not memory-safe:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ynl0ZXMgbWVtb3J5IHg7CmFzc2VtYmx5IHsKICB4IDo9IDB4NDAKfQp4WzB4MjBdID0gMHg0Mjs=)

bytes memory x;

assembly {

x := 0x40

}

x[0x20] = 0x42;

Inline assembly that neither involves any operations that access memory nor assigns to any Solidity variables in memory is automatically considered memory-safe and does not need to be annotated.

**Warning**

It is your responsibility to make sure that the assembly actually satisfies the memory model. If you annotate an assembly block as memory-safe, but violate one of the memory assumptions, this **will** lead to incorrect and undefined behaviour that cannot easily be discovered by testing.

In case you are developing a library that is meant to be compatible across multiple versions of Solidity, you can use a special comment to annotate an assembly block as memory-safe:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=Ly8vIEBzb2xpZGl0eSBtZW1vcnktc2FmZS1hc3NlbWJseQphc3NlbWJseSB7CiAgICAuLi4KfQ==)

*/// @solidity memory-safe-assembly*

assembly {

...

}

Note that we will disallow the annotation via comment in a future breaking release; so, if you are not concerned with backwards-compatibility with older compiler versions, prefer using the dialect string.