26 JUNE 2024

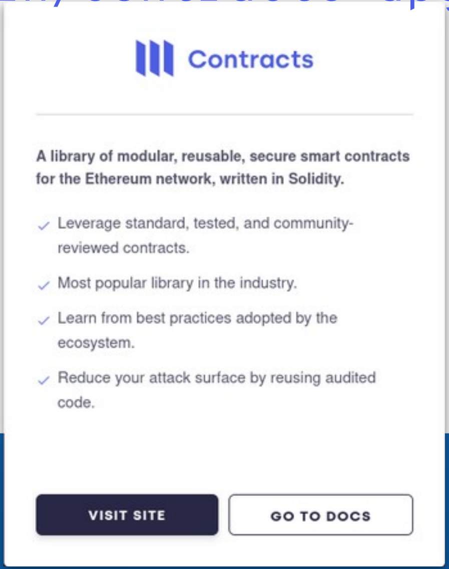
All the things I wish

(native) Solidity would let me do

Contracts

@openzeppelin/contracts@5.5.0

@openzeppelin/contracts-upgradeable@5.5.0



A library of modular, reusable, secure smart contracts for the Ethereum network, written in Solidity.

- ✓ Leverage standard, tested, and community-reviewed contracts.
- ✓ Most popular library in the industry.
- ✓ Learn from best practices adopted by the ecosystem.
- ✓ Reduce your attack surface by reusing audited code.

Designed and implemented:

1. By solidity developers

Designed and implemented:

1. By solidity developers
2. For solidity developers

Designed and implemented:

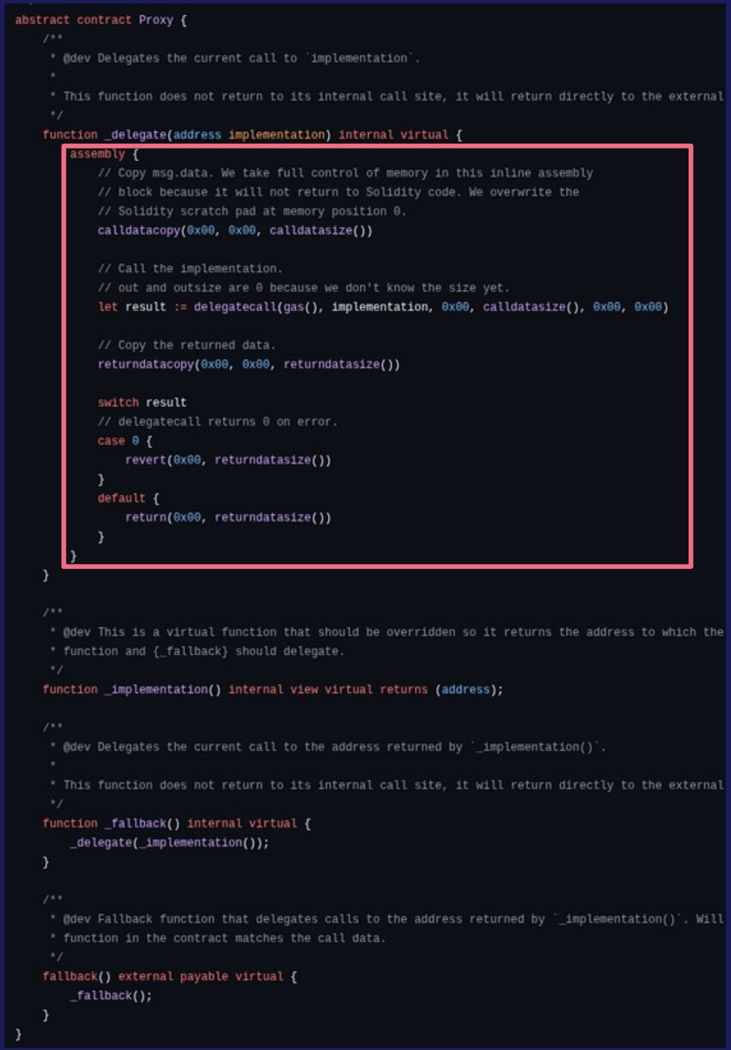
1. By solidity developers
2. For solidity developers
3. In solidity

Our code is not 100% **native** solidity.

@openzeppelin/contracts uses ~400 assembly blocks.

These blocks allow us to leverage features of the EVM that are not (easily/cheaply) available in native solidity !

These blocks reduce the readability/auditability of the code, negatively impacting security.



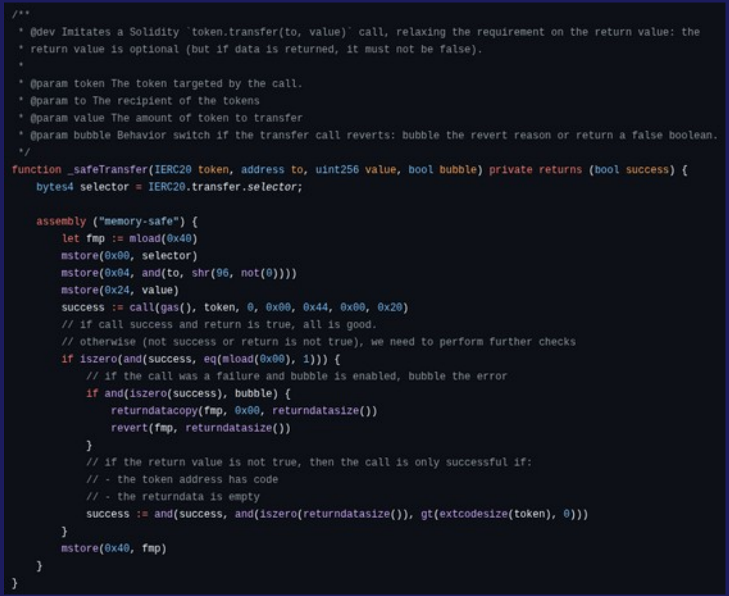
```
abstract contract Proxy {
    /**
     * @dev Delegates the current call to `implementation`.
     *
     * This function does not return to its internal call site, it will return directly to the external
     */
    function _delegate(address implementation) internal virtual {
        assembly {
            // Copy msg.data. We take full control of memory in this inline assembly
            // block because it will not return to Solidity code. We overwrite the
            // Solidity scratch pad at memory position 0.
            calldatacopy(0x00, 0x00, calldatasize())

            // Call the implementation.
            // out and outsize are 0 because we don't know the size yet.
            let result := delegatecall(gas(), implementation, 0x00, calldatasize(), 0x00, 0x00)

            // Copy the returned data.
            returndatacopy(0x00, 0x00, returndatasize())

            switch result
            // delegatecall returns 0 on error.
            case 0 {
                revert(0x00, returndatasize())
            }
            default {
                return(0x00, returndatasize())
            }
        }
    }

    /**
     * @dev This is a virtual function that should be overridden so it returns the address to which the
     * function and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);

    /**
     * @dev Delegates the current call to the address returned by `_implementation()`.
     *
     * This function does not return to its internal call site, it will return directly to the external
     */
    function _fallback() internal virtual {
        _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by `_implementation()`. Will
     * function in the contract matches the call data.
     */
    fallback() external payable virtual {
        _fallback();
    }
}
```

Where do we use assembly?

Low level calls

- Basic call will bubble any revert,
- Try-catch reverts if the target has no code
- Low level calls “address.call(...)” require encoding the parameters, and expand the memory with results

When gas costs are critical, branches must be minimized and memory allocation must be avoided, YUL is king.



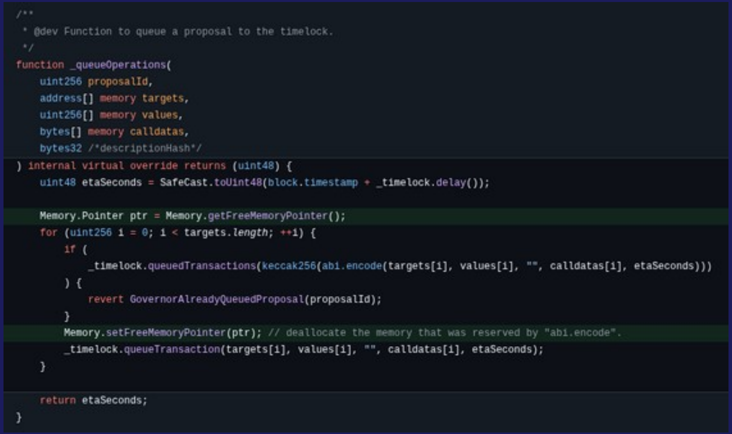
```
/**
 * @dev Imitates a Solidity `token.transfer(to, value)` call, relaxing the requirement on the return value: the
 * return value is optional (but if data is returned, it must not be false).
 *
 * @param token The token targeted by the call.
 * @param to The recipient of the tokens
 * @param value The amount of token to transfer
 * @param bubble Behavior switch if the transfer call reverts: bubble the revert reason or return a false boolean.
 */
function _safeTransfer(IERC20 token, address to, uint256 value, bool bubble) private returns (bool success) {
    bytes4 selector = IERC20.transfer.selector;

    assembly ("memory-safe") {
        let fmp := mload(0x40)
        mstore(0x00, selector)
        mstore(0x04, and(to, shr(96, not(0))))
        mstore(0x24, value)
        success := call(gas(), token, 0, 0x00, 0x44, 0x00, 0x20)
        // if call success and return is true, all is good.
        // otherwise (not success or return is not true), we need to perform further checks
        if iszero(and(success, eq(mload(0x00), 1))) {
            // if the call was a failure and bubble is enabled, bubble the error
            if and(iszero(success), bubble) {
                returndatacopy(fmp, 0x00, returndatasize())
                revert(fmp, returndatasize())
            }
            // if the return value is not true, then the call is only successful if:
            // - the token address has code
            // - the returndata is empty
            success := and(success, and(iszero(returndatasize()), gt(extcodesize(token), 0)))
        }
        mstore(0x40, fmp)
    }
}
```

Deallocating memory

Solidity as a tendency to allocate (reserve) memory, by moving the FMP (free memory pointer).

Memory is never deallocated/cleared.

Memory expansion is expensive. Doing any king of "abi.encode" in a loop can ruin your day if you don't deallocate stuff you don't need.

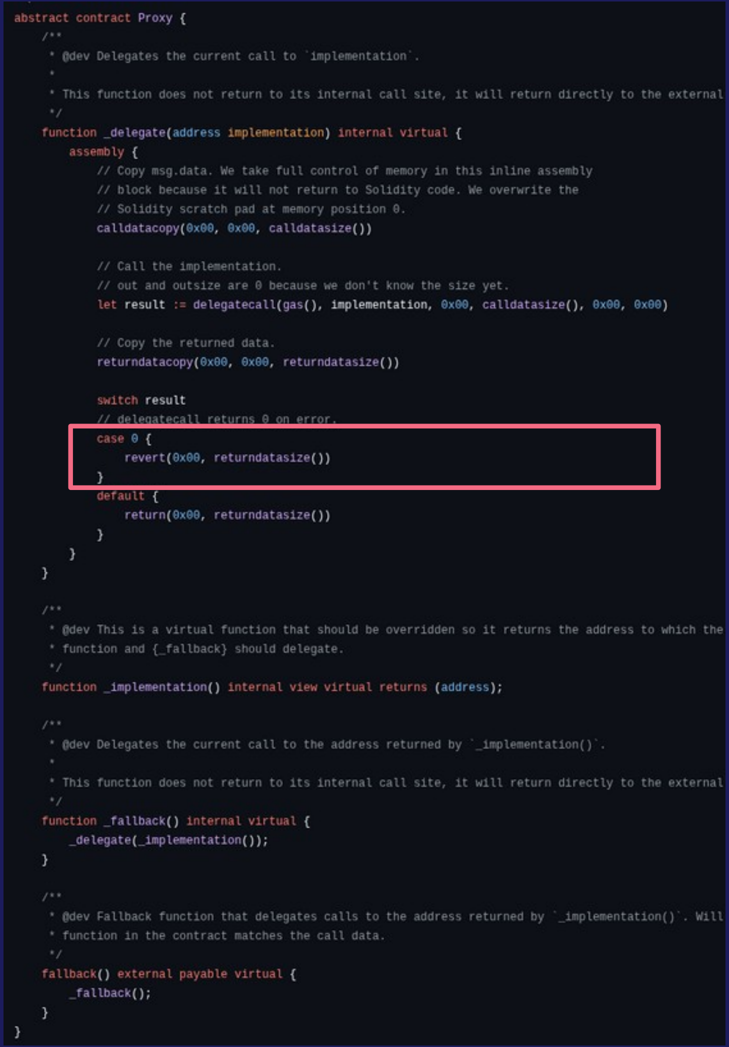


```solidity
/**
 * @dev Function to queue a proposal to the timelock.
 */
function _queueOperations(
    uint256 proposalId,
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    bytes32 /*descriptionHash*/
) internal virtual override returns (uint48) {
    uint48 etaSeconds = SafeCast.toUint48(block.timestamp + _timelock.delay());

    Memory.Pointer ptr = Memory.getFreeMemoryPointer();
    for (uint256 i = 0; i < targets.length; ++i) {
        if (
            _timelock.queuedTransactions(keccak256(abi.encode(targets[i], values[i], "", calldatas[i], etaSeconds)))
        ) {
            revert GovernorAlreadyQueuedProposal(proposalId);
        }
        Memory.setFreeMemoryPointer(ptr); // deallocate the memory that was reserved by "abi.encode".
        _timelock.queueTransaction(targets[i], values[i], "", calldatas[i], etaSeconds);
    }

    return etaSeconds;
}
```

Bubbling reverts

- Basic calls don’t support arbitrary calldata
- Low level calls provide the returndata as a “bytes memory” object
- There is no native way to revert with a raw bytes reason without encapsulation in an “Error(bytes)”

Proxies require bubbling the revert in YUL.



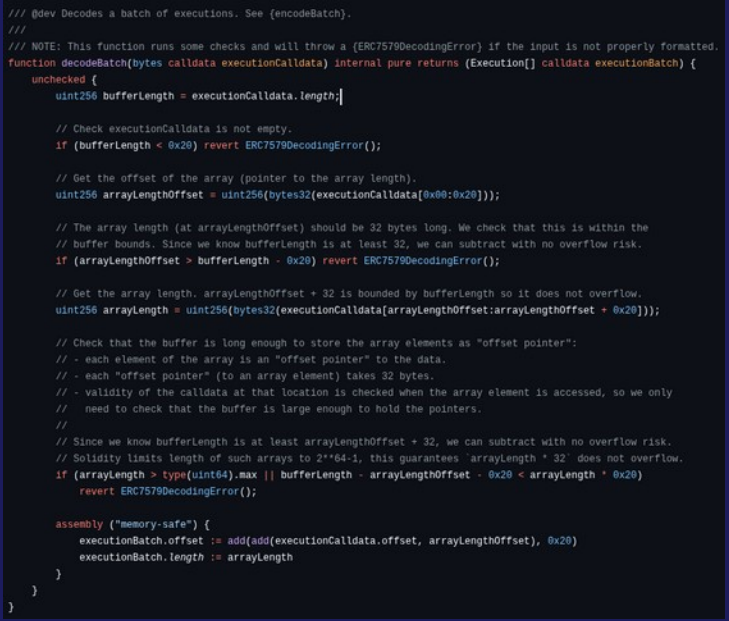
```
abstract contract Proxy {
    /**
     * @dev Delegates the current call to `implementation`.
     *
     * This function does not return to its internal call site, it will return directly to the external
     */
    function _delegate(address implementation) internal virtual {
        assembly {
            // Copy msg.data. We take full control of memory in this inline assembly
            // block because it will not return to Solidity code. We overwrite the
            // Solidity scratch pad at memory position 0.
            calldatacopy(0x00, 0x00, calldatasize())

            // Call the implementation.
            // out and outsize are 0 because we don't know the size yet.
            let result := delegatecall(gas(), implementation, 0x00, calldatasize(), 0x00, 0x00)

            // Copy the returned data.
            returndatacopy(0x00, 0x00, returndatasize())

            switch result
            // delegatecall returns 0 on error.
            case 0 {
                revert(0x00, returndatasize())
            }
            default {
                return(0x00, returndatasize())
            }
        }
    }

    /**
     * @dev This is a virtual function that should be overridden so it returns the address to which the
     * function and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);

    /**
     * @dev Delegates the current call to the address returned by `_implementation()`.
     *
     * This function does not return to its internal call site, it will return directly to the external
     */
    function _fallback() internal virtual {
        _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by `_implementation()`. Will
     * function in the contract matches the call data.
     */
    fallback() external payable virtual {
        _fallback();
    }
}
```

Decode calldata

A calldata object is represented using a pointer with two components (offset and length)

It would be great to be able to do an “abi.decode” that takes calldata input, and returns calldata outputs.

Mostly possible to do in solidity. Still require some YUL to create the “calldata pointer”.



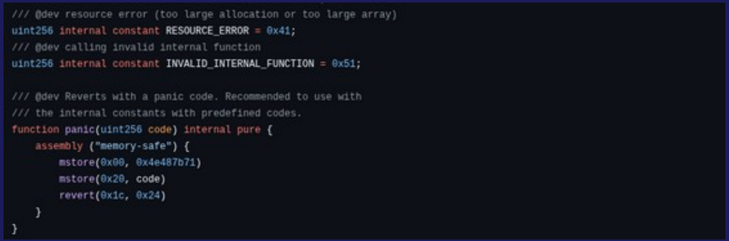
```solidity
/// @dev Decodes a batch of executions. See {encodeBatch}.
///
/// NOTE: This function runs some checks and will throw a {ERC7579DecodingError} if the input is not properly formatted.
function decodeBatch(bytes calldata executionCalldata) internal pure returns (Execution[] calldata executionBatch) {
    unchecked {
        uint256 bufferLength = executionCalldata.length;

        // Check executionCalldata is not empty.
        if (bufferLength < 0x20) revert ERC7579DecodingError();

        // Get the offset of the array (pointer to the array length).
        uint256 arrayLengthOffset = uint256(bytes32(executionCalldata[0x00:0x20]));

        // The array length (at arrayLengthOffset) should be 32 bytes long. We check that this is within the
        // buffer bounds. Since we know bufferLength is at least 32, we can subtract with no overflow risk.
        if (arrayLengthOffset > bufferLength - 0x20) revert ERC7579DecodingError();

        // Get the array length. arrayLengthOffset + 32 is bounded by bufferLength so it does not overflow.
        uint256 arrayLength = uint256(bytes32(executionCalldata[arrayLengthOffset:arrayLengthOffset + 0x20]));

        // Check that the buffer is long enough to store the array elements as "offset pointer":
        // - each element of the array is an "offset pointer" to the data.
        // - each "offset pointer" (to an array element) takes 32 bytes.
        // - validity of the calldata at that location is checked when the array element is accessed, so we only
        //   need to check that the buffer is large enough to hold the pointers.
        //
        // Since we know bufferLength is at least arrayLengthOffset + 32, we can subtract with no overflow risk.
        // Solidity limits length of such arrays to 2**64-1, this guarantees `arrayLength * 32` does not overflow.
        if (arrayLength > type(uint64).max || bufferLength - arrayLengthOffset - 0x20 < arrayLength * 0x20)
            revert ERC7579DecodingError();

        assembly ("memory-safe") {
            executionBatch.offset := add(add(executionCalldata.offset, arrayLengthOffset), 0x20)
            executionBatch.length := arrayLength
        }
    }
}
```

Emit panic code

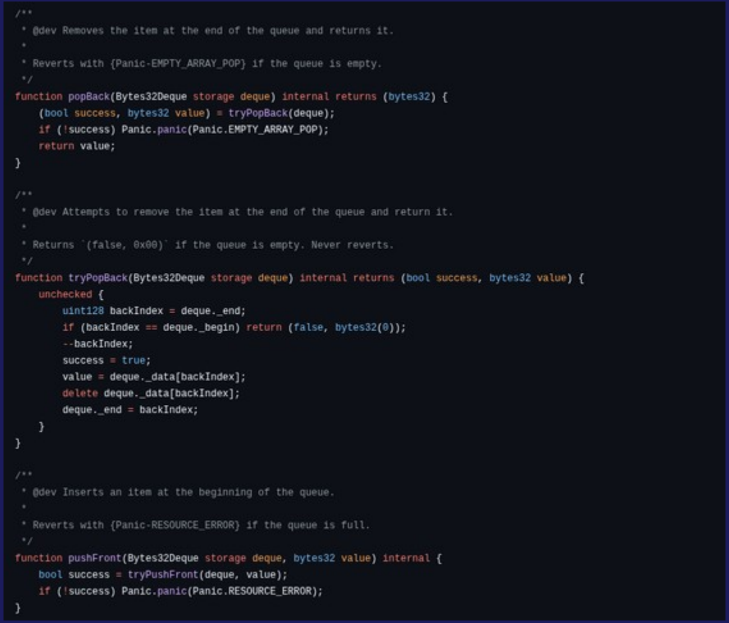
Panic codes are great!

They are “universal custom error” for common/frequent errors (overflow, division by zero, array out-of-bound).

There is no native way to trigger them, which we may want to do when implementing math operations or data structures.



```
/// @dev resource error (too large allocation or too large array)
uint256 internal constant RESOURCE_ERROR = 0x41;
/// @dev calling invalid internal function
uint256 internal constant INVALID_INTERNAL_FUNCTION = 0x51;

/// @dev Reverts with a panic code. Recommended to use with
/// the internal constants with predefined codes.
function panic(uint256 code) internal pure {
    assembly ("memory-safe") {
        mstore(0x00, 0x4e487b71)
        mstore(0x20, code)
        revert(0x1c, 0x24)
    }
}
```



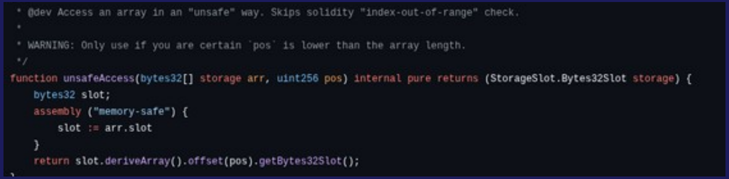
```
/**
 * @dev Removes the item at the end of the queue and returns it.
 *
 * Reverts with {Panic-EMPTY_ARRAY_POP} if the queue is empty.
 */
function popBack(Bytes32Deque storage deque) internal returns (bytes32) {
    (bool success, bytes32 value) = tryPopBack(deque);
    if (!success) Panic.panic(Panic.EMPTY_ARRAY_POP);
    return value;
}

/**
 * @dev Attempts to remove the item at the end of the queue and return it.
 *
 * Returns `(false, 0x00)` if the queue is empty. Never reverts.
 */
function tryPopBack(Bytes32Deque storage deque) internal returns (bool success, bytes32 value) {
    unchecked {
        uint128 backIndex = deque._end;
        if (backIndex == deque._begin) return (false, bytes32(0));
        --backIndex;
        success = true;
        value = deque._data[backIndex];
        delete deque._data[backIndex];
        deque._end = backIndex;
    }
}

/**
 * @dev Inserts an item at the beginning of the queue.
 *
 * Reverts with {Panic-RESOURCE_ERROR} if the queue is full.
 */
function pushFront(Bytes32Deque storage deque, bytes32 value) internal {
    bool success = tryPushFront(deque, value);
    if (!success) Panic.panic(Panic.RESOURCE_ERROR);
}
```

Unchecked accesses

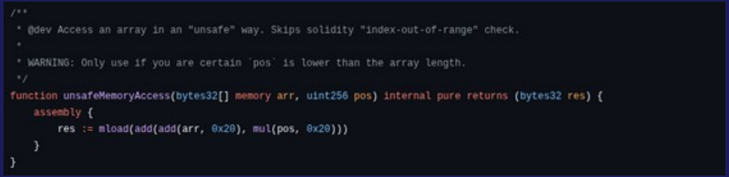
When accessing an array element, solidity will check that the index is within bounds. That check includes a read (possibly from storage) and a branch.

In some cases we know the index is in bound.

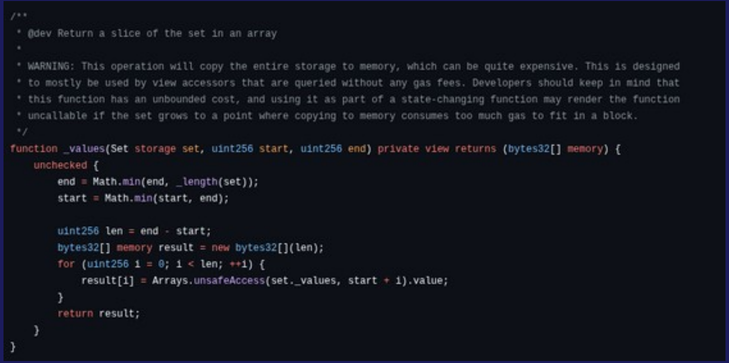
There is no way to skip the check in order to save gas. "unchecked" does not remove this check.



```solidity
 * @dev Access an array in an "unsafe" way. Skips solidity "index-out-of-range" check.
 *
 * WARNING: Only use if you are certain `pos` is lower than the array length.
 */
function unsafeAccess(bytes32[] storage arr, uint256 pos) internal pure returns (StorageSlot.Bytes32Slot storage) {
    bytes32 slot;
    assembly ("memory-safe") {
        slot := arr.slot
    }
    return slot.deriveArray().offset(pos).getBytes32Slot();
```



```solidity
/**
 * @dev Access an array in an "unsafe" way. Skips solidity "index-out-of-range" check.
 *
 * WARNING: Only use if you are certain `pos` is lower than the array length.
 */
function unsafeMemoryAccess(bytes32[] memory arr, uint256 pos) internal pure returns (bytes32 res) {
    assembly {
        res := mload(add(add(arr, 0x20), mul(pos, 0x20)))
    }
}
```



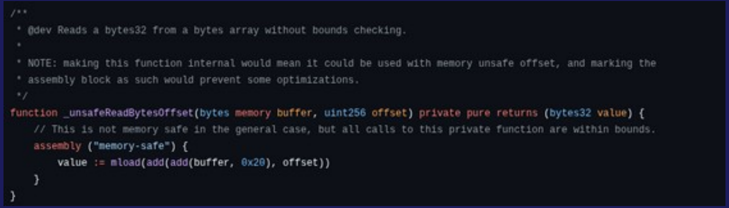
```solidity
/**
 * @dev Return a slice of the set in an array
 *
 * WARNING: This operation will copy the entire storage to memory, which can be quite expensive. This is designed
 * to mostly be used by view accessors that are queried without any gas fees. Developers should keep in mind that
 * this function has an unbounded cost, and using it as part of a state-changing function may render the function
 * uncallable if the set grows to a point where copying to memory consumes too much gas to fit in a block.
 */
function _values(Set storage set, uint256 start, uint256 end) private view returns (bytes32[] memory) {
    unchecked {
        end = Math.min(end, _length(set));
        start = Math.min(start, end);

        uint256 len = end - start;
        bytes32[] memory result = new bytes32[](len);
        for (uint256 i = 0; i < len; ++i) {
            result[i] = Arrays.unsafeAccess(set._values, start + i).value;
        }
        return result;
    }
}
```

Chunks of bytes

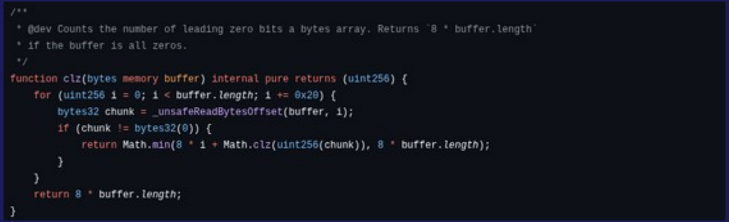
Solidity allows you to access “bytes” and “string” object byte by byte (from memory or storage).

Accessing more than one byte is supported when reading from calldata

If you want to read more than one byte at a time, you need to do it in YUL.



```
/**
 * @dev Reads a bytes32 from a bytes array without bounds checking.
 *
 * NOTE: making this function internal would mean it could be used with memory unsafe offset, and marking the
 * assembly block as such would prevent some optimizations.
 */
function _unsafeReadBytesOffset(bytes memory buffer, uint256 offset) private pure returns (bytes32 value) {
    // This is not memory safe in the general case, but all calls to this private function are within bounds.
    assembly ("memory-safe") {
        value := mload(add(add(buffer, 0x20), offset))
    }
}
```



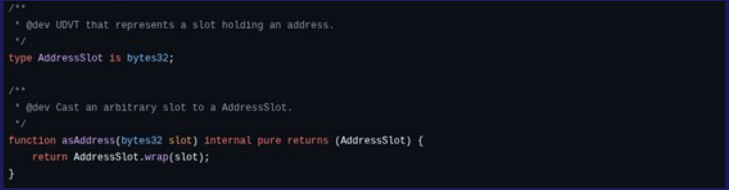
```
/**
 * @dev Counts the number of leading zero bits a bytes array. Returns `8 * buffer.length`
 * if the buffer is all zeros.
 */
function clz(bytes memory buffer) internal pure returns (uint256) {
    for (uint256 i = 0; i < buffer.length; i += 0x20) {
        bytes32 chunk = _unsafeReadBytesOffset(buffer, i);
        if (chunk != bytes32(0)) {
            return Math.min(8 * i + Math.clz(uint256(chunk)), 8 * buffer.length);
        }
    }
    return 8 * buffer.length;
}
```

Transient structures

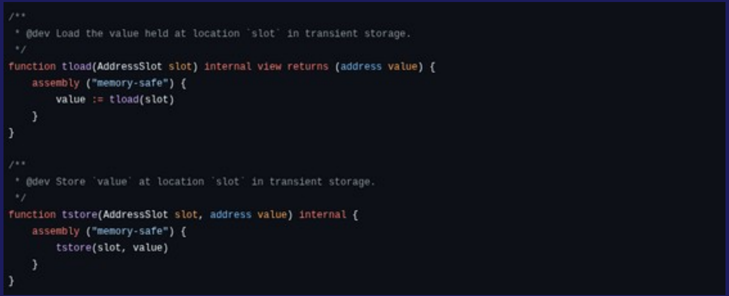
Since version 0.8.28, solidity supports **value type** transient state variable.

Non-value types (including mappings, strings, …) are NOT supported. Application such as Uniswap v4 rely on these.

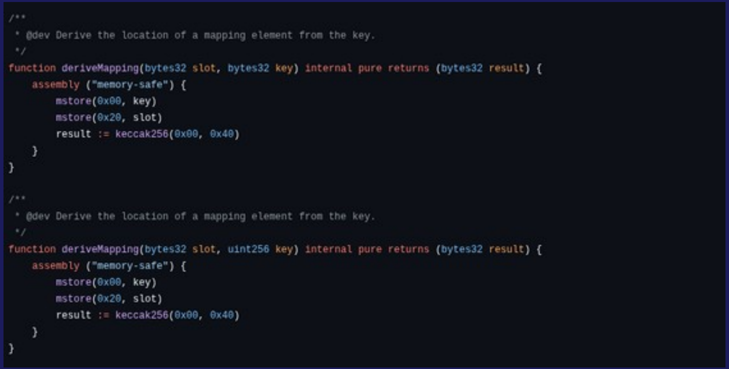
Combining our SlotDerivation and TransientSlot libraries, we can implement transient mappings or array.



```
/**
 * @dev UDVT that represents a slot holding an address.
 */
type AddressSlot is bytes32;

/**
 * @dev Cast an arbitrary slot to a AddressSlot.
 */
function asAddress(bytes32 slot) internal pure returns (AddressSlot) {
    return AddressSlot.wrap(slot);
}
```



```
/**
 * @dev Load the value held at location `slot` in transient storage.
 */
function tload(AddressSlot slot) internal view returns (address value) {
    assembly ("memory-safe") {
        value := tload(slot)
    }
}

/**
 * @dev Store `value` at location `slot` in transient storage.
 */
function tstore(AddressSlot slot, address value) internal {
    assembly ("memory-safe") {
        tstore(slot, value)
    }
}
```



```
/**
 * @dev Derive the location of a mapping element from the key.
 */
function deriveMapping(bytes32 slot, bytes32 key) internal pure returns (bytes32 result) {
    assembly ("memory-safe") {
        mstore(0x00, key)
        mstore(0x20, slot)
        result := keccak256(0x00, 0x40)
    }
}

/**
 * @dev Derive the location of a mapping element from the key.
 */
function deriveMapping(bytes32 slot, uint256 key) internal pure returns (bytes32 result) {
    assembly ("memory-safe") {
        mstore(0x00, key)
        mstore(0x20, slot)
        result := keccak256(0x00, 0x40)
    }
}
```

How hard would it be?

“Trivial”

- Panic codes
- Bubble reverts
- Transient structures

“Easy”

- Calldata decode
- Unchecked accesses
- Chunks of bytes

Hard!

- Deallocate memory
- Low level calls

Can solidity core help?

If OZ can do some of these in YUL, the solidity code "standard library" should be able to solve some of these.

Q&A