Aug 12, 2023 5 min read

Solidity Coding Standards

Updated: Aug 16, 2023

The purpose of this article is not to rehash the official Solidity Style Guide, which you should read. Rather, it is to document the common deviations from the style guide that come up in code reviews or audits. Some of the items here are not in the style guide, but are common stylistic Solidity developer mistakes.

First two lines

1. Include SPDX-License-Identifier

Of course your code will compile without it, but you'll get a warning, so just make the warning go away.

2. Fix the solidity pragma unless writing a library

You've probably seen pragmas such as the following:

```
pragma solidity ^0.8.0;
```

and

```
pragma solidity 0.8.21;
```

Which one should you use and when? If you are the one compiling and deploying the contract, you know the version of Solidity you are compiling with so for the sake of clarity, you should fix the Solidity version to the compiler you are using.

On the other hand, if you are creating a library for other people to extend, such as what OpenZeppelin and Solady do, you should not fix the pragma because you don't know which compiler version the end user will be using.

Imports

3. Explicitly set the library version in the import statement

Instead of doing this:

```
import "@openzepplin/contracts/token/ERC20/ERC20.sol";
```

Do this:

```
import "@openzeppelin/contracts@4.9.3/token/ERC20/ERC20.sol";
```

You can get the latest version by clicking the branch dropdown on the left side of github and clicking tags, then picking the latest release. Use the latest clean (non-rc, i.e. non-release-candidate) version.

If you do not version the import and the underlying library updates, it is possible your code will no longer compile or behave unexpectedly.

4. Use named imports instead of importing the entire namespace

Instead of doing this

```
import "@openzeppelin/contracts@4.9.3/token/ERC20/ERC20.sol";
```

do this

```
import {ERC20} from
"@openzeppelin/contracts@4.9.3/token/ERC20/ERC20.sol";
```

If multiple contracts or libraries are defined inside the import file, you will pollute the namespace. This will result in dead code if the compiler optimizer does not remove it (which you should not rely on).

5. Delete unused imports

If you use a smart contract security tool like Slither, this will be caught automatically. But be sure to remove these. Don't be afraid to delete code.

Contract Level

6. Apply contract-level natspec

The point of natspec (natural language specification) is to provide an easily human-readable inline documentation.

An example natspec for a contract is shown below.

```
/// @title Liquidity token for Foo protocol
/// @author Foo Incorporated
/// @notice Notes for non-technical readers/
/// @dev Notes for people who understand Solidity
contract LiquidityToken {

}
```

7. Lay out the contract structure per the style guide

The functions should be sorted by "externality" first then "state-changingness" second.

They should be ordered as follows: receive and fallback functions (if applicable), external functions, public functions, internal functions, and private functions.
Within those groups, the payable functions go on top, then non-payable, then view, then pure.

```
contract ProperLayout {

    // type declarations, e.g. using Address for address
    // state vars
    address internal owner;
    uint256 internal _stateVar;
    uint256 internal _starteVar2;

    // events
    event Foo();
    event Bar(address indexed sender);

    // errors
    error NotOwner();
    error FooError();
    error BarError();

    // modifiers
    modifier onlyOwner() {
        if (msg.sender != owner) {
            revert NotOwner();
        }
        _;
    }

    // functions
    constructor() {

    }

    receive() external payable {

    }

    falback() external payable {

    }

    // functions are first grouped by
    // - external
    // - public
    // - internal
    // - private
    // note how the external functions "descend" in order of how much
they can modify or interact with the state
    function foo() external payable {

    }

    function bar() external {

    }

    function baz() external view {

    }

    function qux() external pure {

    }

    // public functions
    function fred() public {

    }
```

```
        function bob() public view {

        }

        // internal functions
        // internal view functions
        // internal pure functions
        // private functions
        // private view functions
        // private pure functions
    }
```

Constants

8. Replace magic numbers with constants

If you see the number 100 just sitting in the code, what is it? 100 percent? 100 basis points?

Generally, numbers should be written as a constant at the top of the contract.

9. If numbers are used to measure ether or time, use the solidity keywords

Instead of writing

```
uint256 secondsPerDay = 60 * 60 * 24;
```

do

```
1 days;
```

instead of writing

```
require(msg.value == 10**18 / 10, "must send 0.1 ether");
```

do

```
require(msg.value == 0.1 ether, "must send 0.1 ether");
```

10. Use underscores to make large numbers more readable

Instead of doing this

```
uint256 private constant BASIS_POINTS_DENOMINATOR = 10000
```

do this

```
uint256 private constant BASIS_POINTS_DENOMINATOR = 10_000
```

Functions

11. Remove the virtual modifier from functions that will not be overridden

The virtual modifier means “overridable by a child contract.” But if you know you will not be overriding the function (because you are the deployer), then this modifier is superfluous. Just delete it.

12. Put function modifiers in the correct order visibility, mutability, virtual, override custom modifier

The following is correct

```
// visibility (payability), [virtual], [override], [custom]
function foo() public payable onlyAdmin {

}

function bar() internal view virtual override onlyAdmin {

}
```

13. Use natspec properly

Sometimes referred to as "solidity comment style," its proper name is natspec:

The rules are similar to the contract natspec, except that we also specify params based on the function arguments and what gets returned.

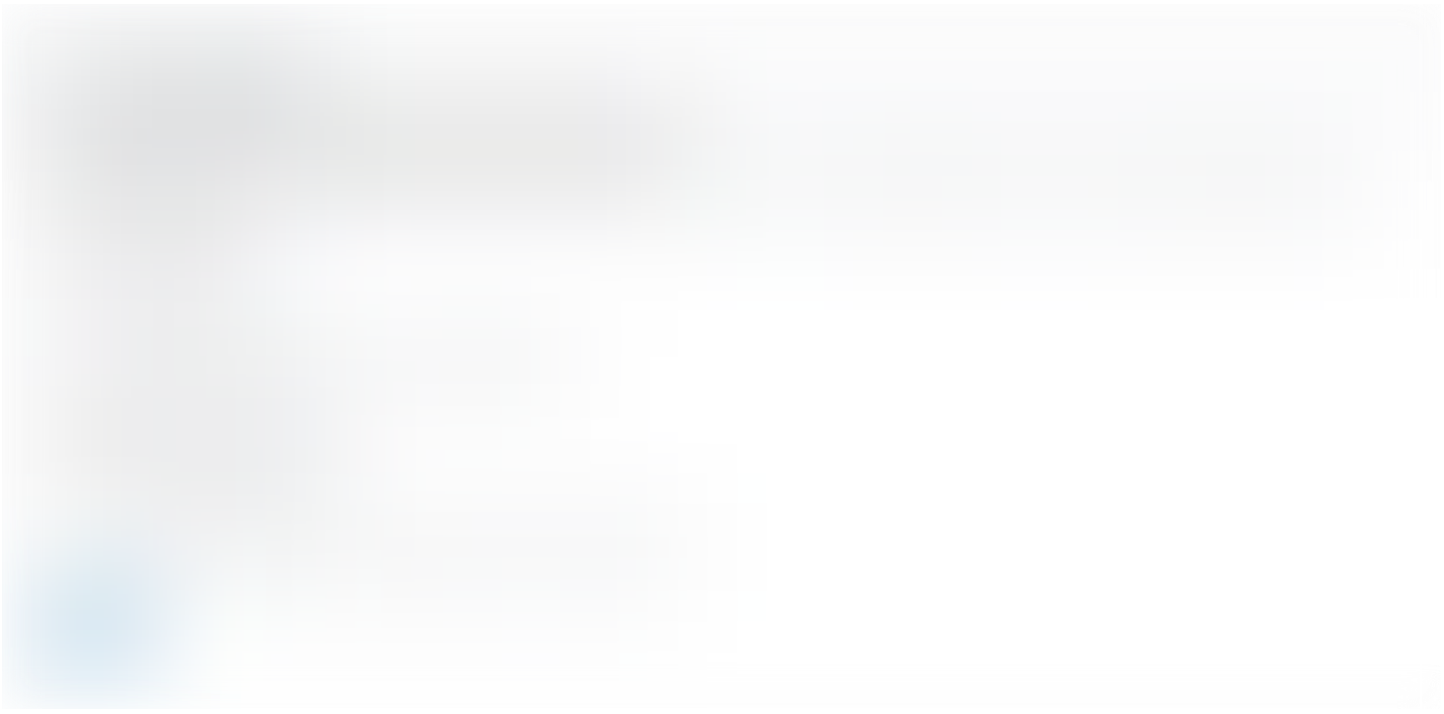
This can be a good way to describe argument names without using long argument variables.

```
/// @notice Deposit ERC20 tokens
/// @dev emits a Deposit event
/// @dev reverts if the token is not allowlisted
/// @dev reverts if the contract is not approved by the ERC20
/// @param token The address of the ERC20 token to be deposited
/// @param amount The amount of ERC20 tokens to deposit
/// @returns the amount of liquidity tokens the user receives
function deposit(address token, uint256 amount) public returns
(uint256) {

}
```

```
// If the contract inherits functions, you can also inherit their
natspec
/// @inheritdoc Lendable
function calculateAccumulatedInterest(address token, uint256 since)
public override view returns (uint256 interest) {

}
```

For the dev parameters, it's good to notify what kind of state changes it can do, for example emitting an event, sending ether, selfdestructing, etc.

The notice and param natspec are read by Etherscan.



You can see where Etherscan got that information from in the following screenshot of the code.

General cleanliness

14. Remove commented out code

This should be self-explanatory. If code is commented out, it’s just clutter.

15. Think carefully about variable names

Naming things is one of the harder aspects of writing good code, but it will do wonders for readability.

Some tips:

- Avoid “generic nouns” like “user.” Be more precise, for example “admin”, “buyer”, “seller”.
- The word “data” is usually an indicator of imprecision. Instead of “userData” do “userAccount”.
- Don’t use two different nouns to apply to the same real-world entity. For example, if “depositor” and “liquidityProvider” refer to the same entity in the real world, just stick to one term, don’t use both in the code.
- Include units in variable names. Instead of “interestRate” do “interestRatesBasisPoints” or “feeInWei”.
- State changing functions should have a verb in the name.
- Be consistent about the use of underscores to distinguish internal variables and functions vs function arguments that overshadow state variables. If prepending a variable with an underscore means “internal” ensure that it isn’t used to mean something else in another context, for example, function arguments that have the same name as a state variable.
- using “get” for viewing data and “set” for changing data is a widely followed programming convention. Consider incorporating it.
- After you finish writing your code, step away from the computer, then come back in 15 minutes and ask yourself for each variable and function name if you were as precise as possible. This deliberate effort will do more good for you than any checklist can provide, since you know the intent of the codebase better than anyone.

Additional Tricks for organizing large codebases

- If you have a lot of storage variables, you can define all the storage variables in a single contract, then inherit from that contract to gain access to those storage variables.
- If your functions require a significant amount of parameters, use a struct to pass the information around.
- If you need a lot of imports, you can import all the files and types into one solidity file, then import that file (you would need to intentionally break the rule about named imports).
- Use libraries to group functions of the same category together and make files smaller.

Organizing large codebases is an art. The best way to learn it is to study codebases of large established projects.

Learn More with RareSkills

This checklist is used in our advanced solidity bootcamp for code reviews.