Sep 7, 2023 49 min read

The RareSkills Book of Solidity Gas Optimization: 80+ Tips

Updated: 2 days ago

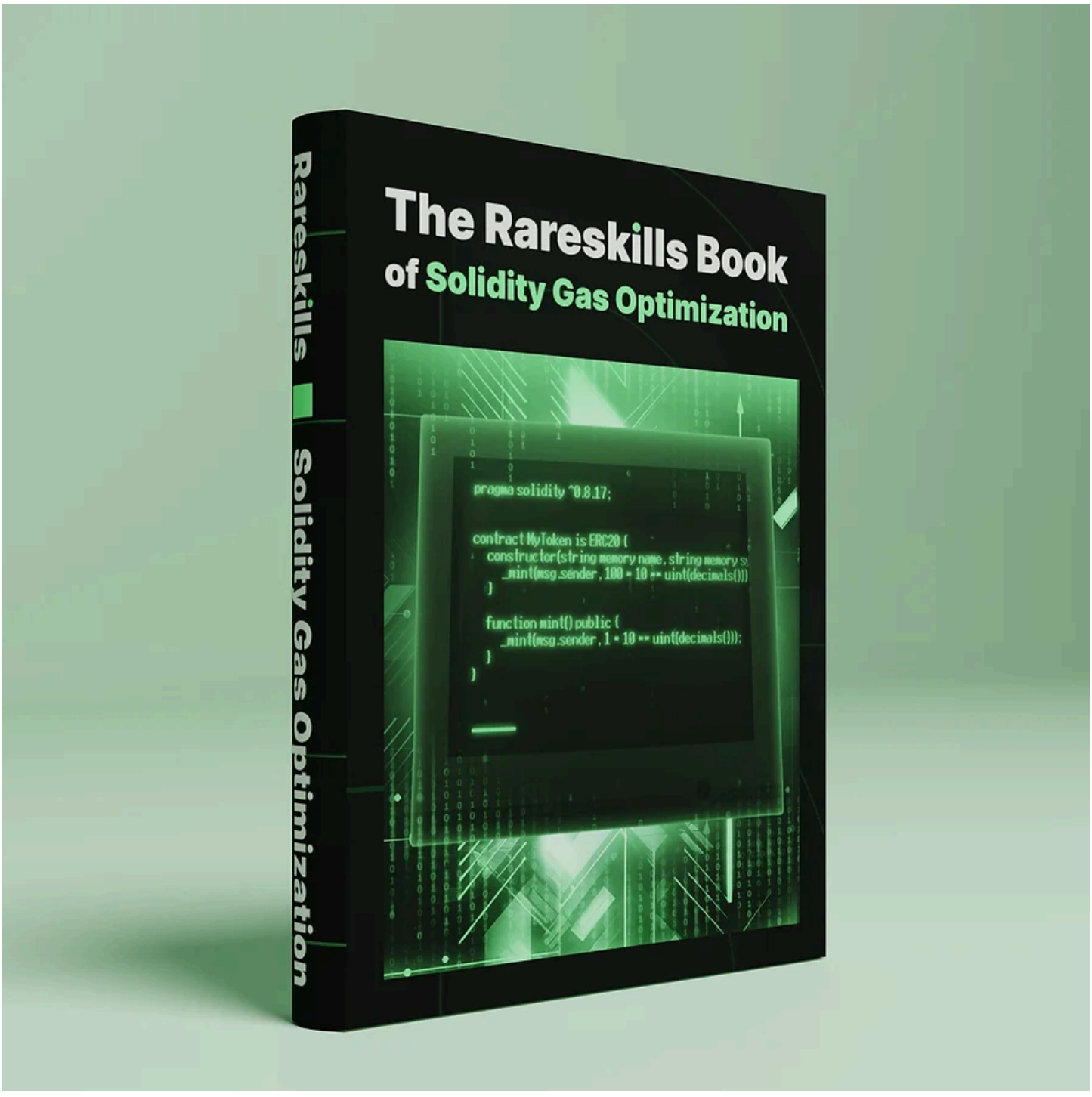

TABLE OF CONTENTS

Solidity Gas Optimization

Gas optimization in Ethereum is re-writing Solidity code to accomplish the same business logic while consuming fewer gas units in the Ethereum Virtual Machine (EVM).

Clocking in at over 11,000 words, not including source code, this article is the most complete treatment of gas optimization available.

To fully understand the tricks in this tutorial, you'll need to understand how the EVM works, which you can learn by taking our Gas Optimization Course, Yul Course, and practicing Huff Puzzles.

However, if you simply want to know what areas of the code to target for possible gas optimizations, this article gives you a lot of areas to look.

Authorship

RareSkills researchers Michael Amadi (LinkedIn, Twitter) and Jesse Raymond (LinkedIn, Twitter) significantly contributed to this work.

Gas optimization tricks do not always work

Some gas optimization tricks only work in a certain context. For example, intuitively, it would seem that

```
if (!cond) {
    // branch False
}
else {
    // branch True
}
```

is less efficient than

```
if (cond) {
    // branch True
}
else {
    // branch False
}
```

because extra opcodes are spent inverting the condition. Counterintuitively, there are many cases where this optimization actually increases the cost of the transaction. The solidity compiler can be unpredictable sometimes.
Therefore, you should actually measure the effect of the alternatives before settling on a certain algorithm. Think of some of these tricks are bringing awareness to areas where the compiler may be surprising.

Some tricks that are not universal are marked as such in this document. Gas optimization tricks sometimes depend on what the compiler is doing locally. You should generally test both the optimal version of the code and the non-optimal version to see you actually get an improvement. We will document some surprising cases where what should lead to an optimization actually leads to higher cost.
Second, some of these optimization behavior may change when using the --via-ir option on the Solidity compiler.

Beware of complexity and readability

Gas optimizations usually make code less readable and more complex. A good engineer must make a subjective tradeoff about what optimizations are worth it, and which ones aren't.

Comprehensive treatment of each topic isn't possible here

We cannot explain each optimization in detail, and it isn't really necessary to as there are other online resources. For example, giving a complete, or even substantial treatment of layer 2s and state channels would be outside of scope, and there are other resources online to learn those subjects in detail.

The purpose of this article is to be the most comprehensive list of tricks out there. If a trick feels unfamiliar, it can be a prompt for further self-study. If the header looks like a trick you already know, just skim over that section.

We do not discuss application-specific tricks

There are gas-efficient ways to determine if a number is prime for example, but this is so rarely needed that dedicating space to it would lower the value of this article. Similarly, in our Tornado Cash tutorial, we suggest ways the codebase could be made more efficient, but including that treatment here would not benefit readers as it is too application specific.

1. Most important: avoid zero to one storage writes where possible

Initializing a storage variable is one of the most expensive operations a contract can do.

When a storage variable goes from zero to non-zero, the user must pay 22,100 gas total (20,000 gas for a zero to non-zero write and 2,100 for a cold storage access).
This is why the Openzeppelin reentrancy guard registers functions as active or not with 1 and 2 rather than 0 and 1. It only costs 5,000 gas to alter a storage variable from non-zero to non-zero.

2. Cache storage variables: write and read storage variables exactly once

You will see the following pattern frequently in efficient solidity code. Reading from a storage variable costs at least 100 gas as Solidity does not cache the storage read. Writes are considerably more expensive. Therefore, you should manually cache the variable to do exactly one storage read and exactly one storage write.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Counter1 {
    uint256 public number;

    function increment() public {
        require(number < 10);
        number = number + 1;
    }
}

contract Counter2 {
    uint256 public number;

    function increment() public {
        uint256 _number = number;
        require(_number < 10);
        number = _number + 1;
    }
}
```

The first function reads counter twice, the second code reads it once.

3. Pack related variables

Packing related variables into same slot reduces gas costs by minimizing costly storage related operations.
Manual packing is the most efficient
We store and retrieve two uint80 values in one variable (uint160) by using bit shifting.
This will use only one storage slot and is cheaper when storing or reading the individual values in a single transaction.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract GasSavingExample {
    uint160 public packedVariables;
```

```
        function packVariables(uint80 x, uint80 y) external {
            packedVariables = uint160(x) << 80 | uint160(y);
        }

        function unpackVariables() external view returns (uint80, uint80) {
            uint80 x = uint80(packedVariables >> 80);
            uint80 y = uint80(packedVariables);
            return (x, y);
        }
    }
```

EVM Packing is slightly less efficient

This also uses one slot like the above example, but may be slightly expensive when storing or reading values in a single transaction.

This is because the EVM will do the bit-shifting itself.

```
    contract GasSavingExample2 {
        uint80 public var1;
        uint80 public var2;

        function updateVars(uint80 x, uint80 y) external {
            var1 = x;
            var2 = y;
        }

        function loadVars() external view returns (uint80, uint80) {
            return (var1, var2);
        }
    }
```

No packing is least efficient

This does not use any optimization, and is more expensive when storing or reading values.

Unlike the other examples, this uses two storage slots to store the variables.

```
    contract NonGasSavingExample {
        uint256 public var1;
        uint256 public var2;

        function updateVars(uint256 x, uint256 y) external {
            var1 = x;
            var2 = y;
```

```
    }

    function loadVars() external view returns (uint256, uint256) {
        return (var1, var2);
    }
}
```

4. Pack structs

Packing struct items, like packing related state variables, can help save gas.
(It is important to note that in Solidity, struct members are stored sequentially in the contract's storage, starting from the slot position where they are initialized).

Consider the following examples:

Unpacked Struct
The unpackedStruct has three items which will be stored in three seperate slot. However, if this items were packed, only two slots would be used and this will make reading and writing to the struct's items cheaper.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Unpacked_Struct {
    struct unpackedStruct {
        uint64 time; // Takes one slot - although it  only uses 64 bits
(8 bytes) out of 256 bits (32 bytes).
        uint256 money; // This will take a new slot because it is a
complete 256 bits (32 bytes) value and thus cannot be packed with the
previous value.
        address person; // An address occupies only 160 bits (20
bytes).
    }

    // Starts at slot 0
    unpackedStruct details = unpackedStruct(53_000, 21_000,
address(0xdeadbeef));

    function unpack() external view returns (unpackedStruct memory) {
        return details;
    }
}
```

Packed Struct

We can make the example above use less gas by packing the struct items like this.

```
contract Packed_Struct {
    struct packedStruct {
        uint64 time; // In this case, both `time` (64 bits) and
`person` (160 bits) are packed in the same slot since they can both fit
into 256 bits (32 bytes)
        address person; // Same slot as `time`. Together they occupy
224 bits (28 bytes) out of 256 bits (32 bytes).
        uint256 money; // This will take a new slot because it is a
complete 256 bits (32 bytes) value and thus cannot be packed with the
previous value.
    }

    // Starts at slot 0
    packedStruct details = packedStruct(53_000, address(0xdeadbeef),
21_000);

    function unpack() external view returns (packedStruct memory) {
        return details;
    }
}
```

5. Keep strings smaller than 32 bytes

In Solidity, strings are variable length dynamic data types, meaning their length can change and grow as needed.

If the length is 32 bytes or longer, the slot in which they are defined stores the length of the string * 2 + 1, while their actual data is stored elsewhere (the keccak hash of that slot).

However, if a string is less than 32 bytes, the length * 2 is stored at the least significant byte of it's storage slot and the actual data of the string is stored starting from the most significant byte in the slot in which it is defined.

String example (less than 32 bytes)

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract StringStorage1 {
    // Uses only one slot
```

```
    // slot 0: 0x(len * 2)00...hex of (len * 2)(hex"hello")
    // Has smaller gas cost due to size.
    string public exampleString = "hello";

    function getString() public view returns (string memory) {
        return exampleString;
    }
}
```

String example (greater than 32 bytes)

```
contract StringStorage2 {
    // Length is more than 32 bytes.
    // Slot 0: 0x00...(length*2+1).
    // keccak256(0x00): stores hex representation of "hello"
    // Has increased gas cost due to size.
    string public exampleString = "This is a string that is slightly
over 32 bytes!";

    function getStringLongerThan32bytes() public view returns (string
memory) {
        return exampleString;
    }
}
```

We can put this to test with following foundry test script:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import "forge-std/Test.sol";
import "../src/StringLessThan32Bytes.sol";

contract StringStorageTest is Test {
    StringStorage1 public store1;
    StringStorage2 public store2;

    function setUp() public {
        store1 = new StringStorage1();
        store2 = new StringStorage2();
    }
```

```
    function testStringStorage1() public {
        // test for string less than 32 bytes
        store1.getString();
        bytes32 data = vm.load(address(store1), 0); // slot 0
        emit log_named_bytes32("Full string plus length", data); // the
full string and its length*2 is stored at slot 0, because it is less
than 32 bytes
    }

    function testStringStorage2() public {
        // test for string longer than 32 bytes
        store2.getStringLongerThan32bytes();
        bytes32 length = vm.load(address(store2), 0); // slot 0 stores
the length*2+1
        emit log_named_bytes32("Length of string", length);

        // uncomment to get original length as number
        // emit log_named_uint("Real length of string (no. of bytes)",
uint256(length) / 2);
        // divide by 2 to get the original length

        bytes32 data1 = vm.load(address(store2),
keccak256(abi.encode(0))); // slot keccak256(0)
        emit log_named_bytes32("First string chunk", data1);

        bytes32 data2 = vm.load(address(store2),
bytes32(uint256(keccak256(abi.encode(0))) + 1));
        emit log_named_bytes32("Second string chunk", data2);
    }
}
```

This is the result after running the test.

If we concatenate the hex value of the string (longer than 32 bytes) without the length, we convert it back to the original string (with Python).

If the length of a string is less than 32 bytes, it's also efficient to store it in a bytes32 variable and use assembly to use it when needed.

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.21;
contract EfficientString {
    bytes32 shortString;

    function getShortString() external view returns(string memory) {
        string memory value;

        assembly {
            // get slot 0
            let slot0Value := sload(shortString.slot)

            // to get the byte that holds the length info, we mask it
to rmove the string and divide it by 2 to get the length
            let len := div(and(slot0Value, 0xff), 2)

            // to get string, we mask the slot value to remove the
length// we are sure that it can't take more than a byte because of the
length check in the `storeShortString` function
            let str := and(slot0Value,
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00)

            // store length in memory
            mstore(0x80, len)

            // store string in memory
            mstore(0xa0, str)

            // make `value` reference 0x80 so that solidity does the
returning for us
            value := 0x80// update the free memory pointer
            mstore(0x40, 0xc0)
        }

        return value;
    }

    function storeShortString(string calldata value) external {
        assembly {
            // require that the length is less than 32
```

```
            if gt(value.length, 31) {
                revert(0, 0)
            }

            // multiply the length, so we can store length*2 following
solidity's convention
            let length := mul(value.length, 2)

            // get the string itself
            let str := calldataload(value.offset)

            // or the length and str to get what we need to store in
storage
            let toBeStored := or(str, length)

            // store it in storage
            sstore(shortString.slot, toBeStored)
        }
    }
}
```

The code above can be further optimized but kept this way to make it easier to understand.

6. Variables that are never updated should be immutable or constant

In Solidity, variables which are not intended to be updated should be constant or immutable.

This is because constants and immutable values are embedded directly into the bytecode of the contract which they are defined and does not use storage because of this.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Constants {
    uint256 constant MAX_UINT256 =
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;

    function get_max_value() external pure returns (uint256) {
        return MAX_UINT256;
    }
}
```

```
// This uses more gas than the above contract
contract NoConstants {
    uint256 MAX_UINT256 =
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;

    function get_max_value() external view returns (uint256) {
        return MAX_UINT256;
    }
}
```

This saves a lot of gas as we do not make any storage reads which are costly.

7. Using mappings instead of arrays to avoid length checks

When storing a list or group of items that you wish to organize in a specific order and fetch with a fixed key/index, it's common practice to use an array data structure. This works well, but did you know that a trick can be implemented to save 2,000+ gas on each read using a mapping?

See the example below

```
/// get(0) gas cost: 4860
contract Array {
    uint256[] a;

    constructor() {
        a.push() = 1;
        a.push() = 2;
        a.push() = 3;
    }

    function get(uint256 index) external view returns(uint256) {
        return a[index];
    }
}

/// get(0) gas cost: 2758
contract Mapping {
    mapping(uint256 => uint256) a;

    constructor() {
```

```
            a[0] = 1;
            a[1] = 2;
            a[2] = 3;
        }

        function get(uint256 index) external view returns(uint256) {
            return a[index];
        }
    }
```

Just by using a mapping, we get a gas saving of 2102 gas. Why? Under the hood when you read the value of an index of an array, solidity adds bytecode that checks that you are reading from a valid index (i.e an index strictly less than the length of the array) else it reverts with a panic error (Panic(0x32) to be precise). This prevents from reading unallocated or worse, allocated storage/memory locations.

Due to the way mappings are (simply a key => value pair), no check like that exists and we are able to read from the a storage slot directly. It's important to note that when using mappings in this manner, your code should ensure that you are not reading an out of bound index of your canonical array.

8. Using unsafeAccess on arrays to avoid redundant length checks

An alternative to using mappings to avoid the length checks that solidity does when reading from arrays (while still using arrays), is using the unsafeAccess function in Openzeppelin's Arrays.sol library. This allows developers to directly access values of any given index of an array while skipping the length overflow check. It's still important to only use this if you are sure that indexes parsed into the function cannot exceed the length of the array parsed in.

9. Use bitmaps instead of bools when a significant amount of booleans are used

A common pattern, especially in airdrops, is to mark an address as "already used" when claiming the airdrop or NFT mint.

However, since it only takes one bit to store this information, and each slot is 256 bits, that means one can store a 256 flags/booleans with one storage slot.

You can learn more about this technique from these resources:
Video tutorial by a student at RareSkills
Bitmap presale tutorial

10. Use SSTORE2 or SSTORE3 to store a lot of data

SSTORE

SSTORE is an EVM opcode that allows us to store persistent data on a key value basis . As everything in EVM, a key and value are both 32 bytes values.
Costs of writing(SSTORE) and reading(SLOAD) are very expensive in terms of gas spent. Writing 32 bytes costs 22,100 gas, which translates to about 690 gas per bytes. On the other hand, writing a smart contract's bytecode costs 200 gas per bytes.

SSTORE2

SSTORE2 is a unique concept in a way that it uses a contract's bytecode to write and store data. To achieve this we use bytecode's inherent property of immutability.
Some properties of SSTORE2:

- We can write only once. Effectively using CREATE instead of SSTORE.
- To read, instead of using SLOAD, we now call EXTCODECOPY on the deployed address where the particular data is stored as bytecode.
- Writing data becomes significantly cheaper when more and more data needs to be stored.

Example:
Writing data
Our goal is to store a specific data (in bytes format) as the contract's bytecode.
To achieve this,We need to do 2 things:-

1. Copy our data to memory first, as EVM then takes this data from memory and store it as runtime code. You can learn more in our article about contract creation code.
2. Return and store the newly deployed contract address for future use.
   - We add the contract code size in place of the four zeroes(0000) between 61 and 80 in the below code
     0x61000080600a3d393df300. Hence if code size is 65, it will become
     0x61004180600a3d393df300(0x0041 = 65)
   - This bytecode is responsible for step 1 we mentioned.
   - Now we return the newly deployed address for step 2.

Final contract bytecode = 00 + data (00 = STOP is prepended to ensure the bytecode cannot be executed by calling the address mistakenly)

Reading data

- To get the relevant data , you need the address where you stored the data.
- We revert if code size is = 0 for obvious reasons.
- Now we simply return the contract's bytecode from the relevant starting position which is after 1 bytes(remember first byte is STOP OPCODE(0x00)).

Additional Information for the curious:

- We can also use pre-deterministic address using CREATE2 to calculate the pointer address off chain or on chain without relying on storing the pointer.

Ref: solady

SSTORE3

To understand SSTORE3, first let's recap an important property of SSTORE2.

- The newly deployed address is dependent on the data we intend to store.

Write data

SSTORE3 implements a design such that the newly deployed address is independent of our provided data. The provided data is first stored in storage using SSTORE.
Then we pass a constant INIT_CODE as data in CREATE2 which internally reads the provided data stored in storage to deploy it as code.
This design choice enables us to efficiently calculate the pointer address of our data just by providing the salt(which can be less than 20 bytes). Thus enabling us to pack our pointer with other variables, thereby reducing storage costs.

Read data

Try to imagine how we could be reading the data.

- **Answer is we can easily compute the deployed address just by providing salt.**
- **Then after we receive the pointer address, use the same EXTCODECOPY opcode to get the required data.**

To summarize:

- ***SSTORE2*** **is helpful in cases where write operations are rare, and large read operations are frequent (and pointer > 14 bytes)**
- ***SSTORE3*** **is better when you write very rarely, but read very often. (and pointer < 14 bytes)**

Credit to Philogy for SSTORE3.

11. Use storage pointers instead of memory where appropriate

In Solidity, storage pointers are variables that reference a location in storage of a contract. They are not exactly the same as pointers in languages like C/C++.
It is helpful to know how to use storage pointers efficiently to avoid unnecessary storage reads and perform gas-efficient storage updates.

Here's an example showing where storage pointers can be helpful.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract StoragePointerUnOptimized {
    struct User {
        uint256 id;
        string name;
        uint256 lastSeen;
    }

    constructor() {
        users[0] = User(0, "John Doe", block.timestamp);
    }
```

```
    mapping(uint256 => User) public users;

    function returnLastSeenSecondsAgo(uint256 _id) public view returns
(uint256) {
        User memory _user = users[_id];
        uint256 lastSeen = block.timestamp - _user.lastSeen;
        return lastSeen;
    }
}
```

Above, we have a function that returns the last seen of a user at a given index. It gets the lastSeen value and subtracts that from the current block.timestamp. Then we copy the whole struct into memory and get the lastSeen which we use in calculating the last seen in seconds ago. This method works well but is not so efficient, this is because we are copying all of the struct from storage into memory including variables we don't need. Only if there was a way to only read from the lastSeen storage slot (without assembly). That's where storage pointers come in.

```
// This results in approximately 5,000 gas savings compared to the
previous version.
contract StoragePointerOptimized {
    struct User {
        uint256 id;
        string name;
        uint256 lastSeen;
    }

    constructor() {
        users[0] = User(0, "John Doe", block.timestamp);
    }

    mapping(uint256 => User) public users;

    function returnLastSeenSecondsAgoOptimized(uint256 _id) public view
returns (uint256) {
        User storage _user = users[_id];
        uint256 lastSeen = block.timestamp - _user.lastSeen;
        return lastSeen;
    }
}
```

"The above implementation results in approximately 5,000 gas savings compared to the first version". Why so, the only change here was changing memory to storage and we were told that anything storage is expensive and should be avoided?

Here we store the storage pointer for users[_id] in a fixed sized variable on the stack (the pointer of a struct is basically the storage slot of the start of the struct, in this case, this will be the storage slot of user[_id].id). Since storage pointers are lazy (meaning they only act(read or write) when called or referenced). Next we only access the lastSeen key of the struct. This way we make a single storage load then store it on the stack, instead of 3 or possibly more storage loads and a memory store before taking a small chunk from memory unto the stack.

Note: When using storage pointers, it's important to be careful not to reference dangling pointers. (Here is a video tutorial on dangling pointers by one of RareSkills' instructors).

12. Avoid having ERC20 token balances go to zero, always keep a small amount

This is related to the avoiding zero writes section above, but it's worth calling out separately because the implementation is a bit subtle.

If an address is frequently emptying (and reloading) it's account balance, this will lead to a lot of zero to one writes.

13. Count from n to zero instead of counting from zero to n

When setting a storage variable to zero, a refund is given, so the net gas spent on counting will be less if the final state of the storage variable is zero.

14. Timestamps and block numbers in storage do not need to be uint256

A timestamp of size uint48 will work for millions of years into the future. A block number increments once every 12 seconds. This should give you a sense of the size of numbers that are sensible.

Saving Gas On Deployment

1. Use the account nonce to predict the addresses of interdependent smart contracts thereby avoiding storage variables and address setter functions

When using traditional contract deployment, the address of a smart contract can be detrministically computed based on the deployer's address and their nonce.

The LibRLP library from Solady can help us do just that.

Take the following example scenario;
StorageContract only allows Writer to set the storage variable x, which means it needs to know the address of Writer. But for Writer to write to StorageContract, it also needs to know the address of StorageContract.

The below implementation is a naive approach to this problem. It handles it by having a setter function which sets a storage variable after deployment. But storage variables are expensive and we'd rather avoid them.

```
contract StorageContract {
    address immutable public writer;
    uint256 public x;

    constructor(address _writer) {
        writer = _writer;
    }

    function setX(uint256 x_) external {
        require(msg.sender == address(writer), "only writer can set");
        x = x_;
    }
}

contract Writer {
    StorageContract public storageContract;

    // cost: 49291
    function set(uint256 x_) external {
        storageContract.setX(x_);
    }

    function setStorageContract(address _storageContract) external {
        storageContract = StorageContract(_storageContract);
    }
}
```

This costs more both at deployment and at runtime. It involves deploying the Writer, then deploying the StorageContract with the deployed Writer address set as the writer. Then setting Writer's StorageContract variable with the newly created StorageContract. This involves a lot of steps and can be expensive since we store StorageContract in storage. Calling Writer.setX() costs 49k gas.

A more efficient way to do this would be to calculate the address the StorageContract and Writer will be deployed to beforehand and set them in both their constructors.

Here's an example of what this would look;

```
import {LibRLP} from
"https://github.com/vectorized/solady/blob/main/src/utils/LibRLP.sol";

contract StorageContract {
    address immutable public writer;
    uint256 public x;

    constructor(address _writer) {
        writer = _writer;
    }

    // cost: 47158
    function setX(uint256 x_) external {
        require(msg.sender == address(writer), "only writer can set");
        x = x_;
    }
}

contract Writer {
    StorageContract immutable public storageContract;

    constructor(StorageContract _storageContract) {
        storageContract = _storageContract;
    }

    function set(uint256 x_) external {
        storageContract.setX(x_);
    }
}

// one time deployer.
contract BurnerDeployer {
    using LibRLP for address;

    function deploy() public returns(StorageContract storageContract,
```

```
    address writer) {
            StorageContract storageContractComputed =
    StorageContract(address(this).computeAddress(2)); // contracts nonce
    start at 1 and only increment when it creates a contract
            writer = address(new Writer(storageContractComputed)); // first
    creation happens here using nonce = 1
            storageContract = new StorageContract(writer); // second create
    happens here using nonce = 2
            require(storageContract == storageContractComputed, "false
    compute of create1 address"); // sanity check
        }
    }
```

Here, calling Writer.setX() costs 47k gas. We saved 2k+ gas by precomputing the address that StorageContract would be deployed to before deploying it so we could use it when deploying Writer, hence no need for a setter function.

It is not required to use a separate contract to employ this technique, you can do it inside the deployment script instead.

We provide a video tutorial of address prediction done by Philogy if you wish to explore this further.

2. Make constructors payable

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.20;

contract A {}

contract B {
    constructor() payable {}
}
```

Making the constructor payable saved 200 gas on deployment. This is because non-payable functions have an implicit require(msg.value == 0) inserted in them. Additionally, fewer bytecode at deploy time mean less gas cost due to smaller calldata.

There are good reasons to make a regular functions non-payable, but generally a contract is deployed by a privileged address who you can reasonably assume won't send ether. This might not apply if inexperienced users are deploying the contract.

3. Deployment size can be reduced by optimizing the IPFS hash to have more zeros (or using the --no-cbor-metadata compiler option)

We've already explained this in our tutorial about smart contract metadata, but to recap, the Solidity compiler appends 51 bytes of metadata to the actual smart contract code. Since each deployment byte costs 200 gas, removing them can take over 10,000 gas cost off of deployment.

This is not always ideal though as it can affect smart contract verification. Instead, developers can mine for code comments that make the IPFS hash that gets appended have more zeros in it.

4. Use selfdestruct in the constructor if the contract is one-time use

Sometimes, contracts are used to deploy several contracts in one transaction, which necessitates doing it in the constructor.

If the contract's only use is the code in the constructor, then selfdestructing at the end of the operation will save gas.

Although selfdestruct is set for removal in an upcoming hardfork, it will still be supported in the constructor per EIP 6780

5. Understand the trade-offs when choosing between internal functions and modifiers

Modifiers inject its implementation bytecode where it is used while internal functions jump to the location in the runtime code where the its implementation is. This brings certain trade-offs to both options.

- Using modifiers more than once means repetitiveness and increase in size of the runtime code but reduces gas cost because of the absence of jumping to the internal function execution offset and jumping back to continue. This means that if runtime gas cost matter most to you, then modifiers should be your choice but if deployment gas cost and/or reducing the size of the creation code is most important to you then using internal functions will be best.
- However, modifiers have the tradeoff that they can only be executed at the start or end of a functon. This means executing it at the middle of a function wouldn't be directly possible, at least not without internal functions which kill the original purpose. This affects it's flexibility. Internal functions however can be called at any point in a function.

Example showing difference in gas cost using modifiers and an internal function

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

/** deployment gas cost: 195435
```

```
    gas per call:
              restrictedAction1: 28367
              restrictedAction2: 28377
              restrictedAction3: 28411
 */
contract Modifier {
    address owner;
    uint256 val;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function restrictedAction1() external onlyOwner {
        val = 1;
    }

    function restrictedAction2() external onlyOwner {
        val = 2;
    }

    function restrictedAction3() external onlyOwner {
        val = 3;
    }
}



/** deployment gas cost: 159309
    gas per call:
              restrictedAction1: 28391
              restrictedAction2: 28401
              restrictedAction3: 28435
 */
contract InternalFunction {
```

```
        address owner;
        uint256 val;

        constructor() {
            owner = msg.sender;
        }

        function onlyOwner() internal view {
            require(msg.sender == owner);
        }

        function restrictedAction1() external {
            onlyOwner();
            val = 1;
        }

        function restrictedAction2() external {
            onlyOwner();
            val = 2;
        }

        function restrictedAction3() external {
            onlyOwner();
            val = 3;
        }
    }
```

Operation	Deployment	restrictedAction 1	restrictedAction 2	restrictedActio n3
Modifiers	195435	28367	28377	28411
Internal Functions	159309	28391	28401	28435

From the table above, we can see that the contract that uses modifiers cost more than 35k gas more than the contract using internal functions when deploying it due to repetition of the onlyOwner functionality in 3 functions.

During runtime, we can see that each function using modifiers cost a fixed 24 gas less than the functions using internal functions.

6. Use clones or metaproxies when deploying very similar smart contracts that are not called frequently

When deploying multiple similar smart contracts, the gas costs can be high. To reduce these costs, you can use minimal clones or metaproxies which store the address of the implementation contract in their bytecode and interact with it as a proxy.
However, there is a trade-off between the runtime cost and deployment cost of clones. Clones are more expensive to interact with than normal contracts due to the delegatecall they use, so they should only be used when you don't need to interact with them frequently. For example, the Gnosis Safe contract uses clones to reduce deployment costs.
Learn more about how to use clones and metaproxies to reduce the gas costs of deploying smart contracts from our blog posts:

- EIP-1167: Minimal Proxy Standard
- EIP-3448 Metaproxy Clone

7. Admin functions can be payable

We can make admin specific functions payable to save gas, because the compiler won't be checking the callvalue of the function.

This will also make the contract smaller and cheaper to deploy as there will be fewer opcodes in the creation and runtime code.

8. Custom errors are (usually) smaller than require statements

Custom errors are cheaper than require statements with strings because of how custom errors are handled. Solidity stores only the first 4 bytes of the hash of the error signature and returns only that. This means during reverting, only 4 bytes needs to be stored in memory. In the case of string messages in require statements, Solidity has to store(in memory) and revert with at least 64 bytes.

Here's an example below.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract CustomError {
    error InvalidAmount();

    function withdraw(uint256 _amount) external pure {
        if (_amount > 10 ether) revert InvalidAmount();
    }
```

```
    }

    // This uses more gas than the above contract
    contract NoCustomError {
        function withdraw(uint256 _amount) external pure {
            require(_amount <= 10 ether, "Error: Pass in a valid amount");
        }
    }
```

9. Use existing create2 factories instead of deploying your own

The title is self-explanatory. If you need a deterministic address, you can usually re-use a pre-deployed one.

Cross contract calls

1. Use transfer hooks for tokens instead of initiating a transfer from the destination smart contract

Let's say you have contract A which accepts token B (an NFT or an ERC1363 token). The naive workflow is as follows:

1. **msg.sender approves contract A to accept token B**
2. **msg.sender calls contract A to transfer tokens from msg.sender to A**
3. **Contract A then calls token B to do the transfer**
4. **Token B does the transfer, and calls onTokenReceived() in contract A**
5. **Contract A returns a value from onTokenReceived() to token B**
6. **Token B returns execution to contract A**

This is very inefficient. It's better for msg.sender to call contract B to do a transfer which calls the tokenReceived hook in contract A.

Note that:

- **All ERC1155 tokens include a transfer hook**
- **safeTransfer and safeMint in ERC721 have a transfer hook**
- **ERC1363 has transferAndCall**
- **ERC777 has a transfer hook but has been deprecated. Use ERC1363 or ERC1155 instead if you need fungible tokens**

If you need to pass arguments to contract A, simply use the data field and parse that in contract A.

2. Use fallback or receive instead of deposit() when transferring Ether

Similar to above, you can "just transfer" ether to a contract and have it respond to the transfer instead of using a payable function. This of course, depends on the rest of the contract's architecture.

Example Deposit in AAVE

```
contract AddLiquidity{

    receive() external payable {
      IWETH(weth).deposit{msg.value}();
      AAVE.deposit(weth, msg.value, msg.sender, REFERRAL_CODE)
    }
}
```

The fallback function is capable of receiving bytes data which can be parsed with abi.decode. This servers as an alternative to supplying arguments to a deposit function.

3. Use ERC2930 access list transactions when making cross-contract calls to pre-warm storage slots and contract addresses

Access list transactions allow you to prepay the gas costs for some storage and call operations, with a 200 gas discount. This can save gas on further state or storage access, which is paid as a warm access.

If your transaction will make a cross-contract call, you should almost certainly be using access list transactions.

When calling clones or proxies which always involve a cross-contract call via delegatecall, you should make the transaction an access list transaction.

We have a dedicated blog post on this, visit https://www.rareskills.io/post/eip-2930-optional-access-list-ethereum to learn more.

4. Cache calls to external contracts where it makes sense (like caching return data from chainlink oracle)

Caching data is generally recommended to avoid duplication in memory when you want to use the same data > 1 times during a single execution process.

Obvious example is if you need to make multiple operations, say, using ETH price gotten from chainlink. You store the price in memory, instead of making the expensive external call again.

5. Implement multicall in router-like contracts

This is a common feature, such as the Uniswap Router and the Compound Bulker.
If you expect your users to make a sequence of calls, have a contract batch them together using multicall.

6. Avoid contract calls by making the architecture monolithic

Contract calls are expensive, and the best way to save gas on them is by not using them at all. There is a natural tradeoff with this, but having several contracts that talk to each other can sometimes increase gas and complexity rather than manage it.

Design Patterns

1. Use multidelegatecall to batch transactions

Multi-delegatecall helps the msg.sender to call multiple functions within a contract while preserving the env vars like msg.sender and msg.value .

***Note*: Be mindful that since msg.value is persistent, it can lead to issues that the developer need to address while inheriting multi delegatecall in their contract.**

Example of Multi delegatecall is Uniswap's implementation below:

```
function multicall(bytes[] calldata data) public payable override
returns (bytes[] memory results) {
        results = new bytes[](data.length);
        for (uint256 i = 0; i < data.length; i++) {
            (bool success, bytes memory result) =
address(this).delegatecall(data[i]);

            if (!success) {
                // Next 5 lines from
https://ethereum.stackexchange.com/a/83577
                if (result.length < 68) revert();
                assembly {
                    result := add(result, 0x04)
                }
                revert(abi.decode(result, (string)));
            }
```

```
                results[i] = result;
            }
        }
```

2. Use ECDSA signatures instead of merkle trees for allowlists and airdrops

Merkle trees use a considerable amount of calldata and increase in cost with the size of the merkle proof. Generally, using digital signatures is cheaper gas-wise compared to merkle proofs.

3. Use ERC20Permit to batch the approval and transfer step in on transaction

ERC20 Permit has an additional function that accepts digital signatures from a toke holder to increase the approval for another address. This way, the recipient of the approval can submit the permit transaction and the transfer in one transaction. The user granting the permit does not have to pay any gas, and the recipient of the permit can batch the permit and transferFrom transaction into a single transaction.

4. Use L2 message passing for games or other high-throughput, low transaction value applications

Etherorcs was one of the early pioneers of this pattern, so you can look to their Github (linked above) for inspiration. The idea is that assets on Ethereum can be "bridge" (via message passing) to another chain such as Polygon, Optimism, or Arbitrum and the game can be conducted there where transactions are cheap.

5. Use state-channels if applicable

State channels are probably the oldest, but still useable scalability solutions for Ethereum. Unlike L2s, they are application specific. Rather than users commiting their transactions to a chain, they commit assets to a smart contract then share binding signatures with each other as state transitions. When the operation is over, they then commit the final result to the chain.

If one of the participants is dishonest, then an honest participant can use the counterparty's signature to force the smart contract to release their assets.

6. Use voting delegation as a gas saving measure

Our tutorial on ERC20 Votes describes this pattern in more detail. Instead of every token owner voting, only the delegates vote, which net reduces the number of voting transactions.

7. ERC1155 is a cheaper non-fungible token than ERC721

The ERC721 balanceOf function is rarely used in practice but adds a storage overhead whenever a mint and transfer happens. ERC1155 tracks balance per id, and also uses the same balance to track ownership of the id. If the maximum supply for each id is one, then the token becomes non-fungible per each id.

8. Use one ERC1155 or ERC6909 token instead of several ERC20 tokens

This was the original intent of the ERC1155 token. Each individual token behaves like and ERC20, but only one contract needs to be deployed.

The drawback of this approach is that the tokens will not be compatible with most DeFi swapping primitives.

ERC1155 uses callbacks on all of the transfer methods. If this is not desired, ERC6909 can be used instead.

9. The UUPS upgrade pattern is more gas efficient for users than the Transparent Upgradeable Proxy

The transparent upgradeable proxy pattern requires comparing msg.sender to the admin every time a transaction happens. UUPS only does this for the upgrade function.

10. Consider using alternatives to OpenZeppelin

OpenZeppelin is a great and popular smart contract library, but there are other alternatives that are worth considering. These alternatives offer better gas efficiency and have been tested and recommended by developers.

Two examples of such alternatives are Solmate and Solady.

Solmate is a library that provides a number of gas-efficient implementations of common smart contract patterns. Solady is another gas-efficient library that places a strong emphasis on using assembly.

Calldata Optimizations

Ethereum charges 4 gas for a zero byte of calldata and 16 gas for a non-zero byte. This is true during a normal function call and during deployment. Because of this, Solidity optimizers try to use zeros where possible.

1. Use vanity addresses (safely!)

It is cheaper to use vanity addresses with leading zeros, this saves calldata gas cost.

A good example is OpenSea Seaport contract with this address: 0x00000000000000ADc04C56Bf30aC9d3c0aAF14dC.

This will not save gas when calling the address directly. However, if that contract's address is used as an argument to a function, that function call will cost less gas due to having more zeros in the calldata.

This is also true of passing EOAs with a lot of zeros as a function argument – it saves gas for the same reason.

Just be aware that there have been hacks from generating vanity addresses for wallets with insufficiently random private keys. This is not a concert for smart contracts vanity addresses created with finding a salt for create2, because smart contracts do not have private keys.

2. Avoid signed integers in calldata if possible

Because solidity uses two's complement to represent signed integers, calldata for small negative numbers will be largely non-zero. For example, -1 is 0xff..ff in two's complement form and therefore more expensive.

3. Calldata is (usually) cheaper than memory

Loading function inputs or data directly from calldata is cheaper compared to loading from memory. This is because accessing data from calldata involves fewer operations and gas costs. Hence, it is advised to use memory only when the data needs to be modified in the function (calldata cannot be modified).

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;
contract CalldataContract {
    function getDataFromCalldata(bytes calldata data) public pure
returns (bytes memory) {
        return data;
    }
}

contract MemoryContract {
    function getDataFromMemory(bytes memory data) public pure returns
(bytes memory) {
        return data;
```

```
        }
    }
```

4. Consider packing calldata, especially on an L2

Solidity automatically packs storage variables, but the abi encoding for variables that would be packed in storage are not packed in calldata.

This is a rather extreme optimization that leads to higher code complexity, but it is something to consider if a function takes a lot of calldata.

ABI encoding is not efficient for every data representation, some data representations can be encoded more efficiently in an application specific way.

Assembly tricks

You should not assume that writing assembly will automatically lead to more efficient code. We've listed areas where writing assembly usually works better, but you should always test the non-assembly version.

1. Using assembly to revert with an error message

When reverting in solidity code, it is common practice to use a require or revert statement to revert execution with an error message. This can in most cases be further optimized by using assembly to revert with the error message.

Here's an example;

```
/// calling restrictedAction(2) with a non-owner address: 24042
contract SolidityRevert {
    address owner;
    uint256 specialNumber = 1;

    constructor() {
        owner = msg.sender;
    }

    function restrictedAction(uint256 num)  external {
        require(owner == msg.sender, "caller is not owner");
        specialNumber = num;
    }
```

```
    }

    /// calling restrictedAction(2) with a non-owner address: 23734
    contract AssemblyRevert {
        address owner;
        uint256 specialNumber = 1;

        constructor() {
            owner = msg.sender;
        }

        function restrictedAction(uint256 num)  external {
            assembly {
                if sub(caller(), sload(owner.slot)) {
                    mstore(0x00, 0x20) // store offset to where length of
    revert message is stored
                    mstore(0x20, 0x13) // store length (19)
                    mstore(0x40,
    0x63616c6c6572206973206e6f74206f776e657200000000000000000000000000) //
    store hex representation of message
                    revert(0x00, 0x60) // revert with data
                }
            }
            specialNumber = num;
        }
    }
```

From the example above we can see that we get a gas saving of over 300 gas when reverting wth the same error message with assembly against doing so in solidity. This gas savings come from the memory expansion costs and extra type checks the solidity compiler does under the hood.

2. Calling functions via interface incurs memory expansion costs, so use assembly to re-use data already in memory

When calling a function on a contract B from another contract A, it's most convenient to use the interface, create an instance of B with an address and call the function we wish to call. This works very well, but due to how solidity compiles our code, it stores the data to send to contract B in a new memory location thereby expanding memory, sometimes unnecessarily.

With inline assembly, we can optimize our code better and save some gas by using previously used memory locations that we don't need again or (if the calldata contract B expects is less than 64

bytes) in the scratch space to store our calldata.

Here's an example comparing the two;

```
/// 30570
contract Sol {
    function set(address addr, uint256 num) external {
        Callme(addr).setNum(num);
    }
}

/// 30350
contract Assembly {
    function set(address addr, uint256 num) external {
        assembly {
            mstore(0x00, hex"cd16ecbf")
            mstore(0x04, num)

            if iszero(extcodesize(addr)) {
                revert(0x00, 0x00) // revert if address has no code
deployed to it
            }

            let success := call(gas(), addr, 0x00, 0x00, 0x24, 0x00,
0x00)

            if iszero(success) {
                revert(0x00, 0x00)
            }
        }
    }
}


contract Callme {
    uint256 num = 1;

    function setNum(uint256 a) external {
        num = a;
```

```
        }
    }
```

We can see that calling set(uint256) on Assembly costs 220 gas less than it would cost if we used solidity.

Do note that when using inline assembly to make external calls, it's important to check if the address we are calling has code deployed to it using extcodesize(addr) and revert if this returns 0. This is important because calling an address that has no code deployed to it always returns true which can be devastating for our contract logic in most scenarios.

3. Common math operations, like min and max have gas efficient alternatives

Unoptimized

```
function max(uint256 x, uint256 y) public pure returns (uint256 z) {
    z = x > y ? x : y;
}
```

Optimized

```
function max(uint256 x, uint256 y) public pure returns (uint256 z) {
    /// @solidity memory-safe-assembly
    assembly {
        z := xor(x, mul(xor(x, y), gt(y, x)))
    }
}
```

The code above is taken from the math section of the Solady Library, more math operations can be found there. It is worth exploring the library to see what gas efficient operations are available to you.

The reason the above example is more gas efficient is because the ternary operator (and in general, code with conditionals in it) contain conditional jumps in the opcodes, which are more costly.

4. Use SUB or XOR instead of ISZERO(EQ()) to check for inequality (more efficient in certain scenarios)

When using inline assembly to compare the equality of two values (e.g if owner is the same as caller()), It is sometimes more efficient to do

```
if sub(caller, sload(owner.slot)) {
    revert(0x00, 0x00)
}
```

over doing this

```
if eq(caller, sload(owner.slot)) {
    revert(0x00, 0x00)
}
```

xor can accomplish the same thing, but be aware that xor will consider a value with all the bits flipped to be equal also, so make sure that this cannot become an attack vector.

This trick will depend on the compiler version used and the context of the code.

5. Use inline assembly to check for address(0)

Writing contracts in inline assembly is generally considered gas optimized. we can manipulate memory directly and use fewer opcodes instead of leaving it to the Solidity compiler.

Authentication mechanism is one example where using inline assembly is good, like implementing address zero check.
Here's an example below:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract NormalAddressZeroCheck {
    function check(address _caller) public pure returns (bool) {
        require(_caller != address(0x00), "Zero address");
        return true;
    }
}

contract AddressZeroCheckAssembly {
    // Saves about 90 gasfunction checkOptimized(address _caller)
public pure returns (bool) {
        assembly {
            if iszero(_caller) {
                mstore(0x00, 0x20)
                mstore(0x20, 0x0c)
                mstore(0x40,
0x5a65726f2041646472657373000000000000000000000000000000000000000) //
load hex of "Zero Address" to memory
                revert(0x00, 0x60)
            }
        }
```

```
            return true;
        }
    }
```

6. selfbalance is cheaper than address(this).balance (more efficient in certain scenarios)

The solidity code address(this).balance can sometimes be done more efficiently with the selfbalance() function from yul, but be aware the compiler is sometimes smart enough to use this trick under the hood, so test both ways.

7. Use assembly to perform operations on data of size 96 bytes or less: hashing and unindexed data in events

Solidity always writes to memory by expanding it which sometimes is not efficient. We can optimize memory operations on data that are 96 bytes in size or less by utilizing inline-assembly.

Solidity reserves it's first 64 bytes of memory (mem[0x00:0x40]) as scratch space that devs can use to perform any operation with guarantees that it won't be overwritten or read from unexpectedly. The next 32 bytes of memory (mem[0x40:0x60]) is where solidity stores, reads and updates the free memory pointer from. This is how solidity keeps track of the next memory offset to write new data to. The next 32 bytes of memory (mem[0x60:0x80]) is called the zero slot. It is where uninitialized dynamic memory data (bytes memory, string memory, T[] memory(where T is any valid type)) points to. Since these values are uninitialized, solidity advances that the slot they point to (0x60) remain 0x00.

Note: Structs stored in memory even when dynamic (i.e have a dynamic value within themselves), when uninitialized do not point to the zero slot.

Note: Uninitialized dynamic memory data still point to the zero slot even if they're nested within a struct.

If we can utilize the scratch space in performing operations in memory that the compiler would usually expand memory to perform if it did so itself, then we can optimize our code. So we have 64 bytes of cheaper memory to work with now.
The free memory pointer space can also be used as long as we update it before exiting the assembly block too. We can store it on the stack temporarily for this.

Let's see some examples.

- **Using assembly to log up to 96 bytes of unindexed data**

```
contract ExpensiveLogger {
    event BlockData(uint256 blockTimestamp, uint256 blockNumber,
uint256 blockGasLimit);
```

```
    // cost: 26145
    function returnBlockData() external {
        emit BlockData(block.timestamp, block.number, block.gaslimit);
    }
}


contract CheapLogger {
    event BlockData(uint256 blockTimestamp, uint256 blockNumber,
uint256 blockGasLimit);

    // cost: 22790
    function returnBlockData() external {
        assembly {
            mstore(0x00, timestamp())
            mstore(0x20, number())
            mstore(0x40, gaslimit())

            log1(0x00,
                0x60,

0x9ae98f1999f57fc58c1850d34a78f15d31bee81788521909bea49d7f53ed270b //
event hash of BlockData
                )
        }
    }
}
```

The example above shows how we can save almost 2,000 gas by using memory to store the data we wish to emit in the BlockData event.
There is no need to update our free memory pointer here because execution ends right after we emit our event and we never step back into solidity code.

Let's take another example where we would need to update the free memory pointer

- **Using assembly to hash up to 96 bytes of data**

```
contract ExpensiveHasher {
    bytes32 public hash;
    struct Values {
        uint256 a;
        uint256 b;
```

```
        uint256 c;
    }
    Values values;

    // cost: 113155function setOnchainHash(Values calldata _values)
external {
        hash = keccak256(abi.encode(_values));
        values = _values;
    }
}


contract CheapHasher {
    bytes32 public hash;
    struct Values {
        uint256 a;
        uint256 b;
        uint256 c;
    }
    Values values;

    // cost: 112107
    function setOnchainHash(Values calldata _values) external {
        assembly {
            // cache the free memory pointer because we are about to
override it
            let fmp := mload(0x40)

            // use 0x00 to 0x60
            calldatacopy(0x00, 0x04, 0x60)
            sstore(hash.slot, keccak256(0x00, 0x60))

            // restore the cache value of free memory pointer
            mstore(0x40, fmp)
        }

        values = _values;
    }
}
```

In the above example, similar to the first one, we use assembly to store values in the first 96 bytes of memory which saves us 1,000+ gas. Also notice that in this instance, because we still break back into solidity code, we cached and updated our free memory pointer at the start and end of our assembly block. This is to make sure that the solidity compiler's assumptions on what is stored in memory remains compatible.

8. Use assembly to reuse memory space when making more than one external call.

An operation that causes the solidity compiler to expand memory is making external calls. When making external calls the compiler has to encode the function signature of the function it wishes to call on the external contract alongside it's arguments in memory. As we know, solidity does not clear or reuse memory memory so it'll have to store these data in the next free memory pointer which expands memory further.

With inline assembly, we can either use the scratch space and free memory pointer offset to store this data (as above) if the function arguments do not take up more than 96 bytes in memory. Better still, if we are making more than one external call we can reuse the same memory space as the first calls to store the new arguments in memory without expanding memory unnecessarily. Solidity in this scenario would expand memory by as much as the returned data length is. This is because the returned data is stored in memory (in most cases). If the return data is less than 96 bytes, we can use the scratch space to store it to prevent expanding memory.

See the example below;

```
contract Called {
    function add(uint256 a, uint256 b) external pure returns(uint256) {
        return a + b;
    }
}


contract Solidity {
    // cost: 7262
    function call(address calledAddress) external pure returns(uint256)
{
        Called called = Called(calledAddress);
        uint256 res1 = called.add(1, 2);
        uint256 res2 = called.add(3, 4);

        uint256 res = res1 + res2;
        return res;
    }
```

```
    }


    contract Assembly {
        // cost: 5281
        function call(address calledAddress) external view returns(uint256)
    {
            assembly {
                // check that calledAddress has code deployed to it
                if iszero(extcodesize(calledAddress)) {
                    revert(0x00, 0x00)
                }

                // first call
                mstore(0x00, hex"771602f7")
                mstore(0x04, 0x01)
                mstore(0x24, 0x02)
                let success := staticcall(gas(), calledAddress, 0x00, 0x44,
    0x60, 0x20)
                if iszero(success) {
                    revert(0x00, 0x00)
                }
                let res1 := mload(0x60)

                // second call
                mstore(0x04, 0x03)
                mstore(0x24, 0x4)
                success := staticcall(gas(), calledAddress, 0x00, 0x44,
    0x60, 0x20)
                if iszero(success) {
                    revert(0x00, 0x00)
                }
                let res2 := mload(0x60)

                // add results
                let res := add(res1, res2)

                // return data
                mstore(0x60, res)
                return(0x60, 0x20)
```

```
            }
        }
    }
```

We save approximately 2,000 gas by using the scratch space to store the function selector and it's arguments and also reusing the same memory space for the second call while storing the returned data in the zero slot thus not expanding memory.

If the arguments of the external function you wish to call is above 64 bytes and if you are making one external call, it wouldn't save any significant gas writing it in assembly. However, if making more than one call. You can still save gas by reusing the same memory slot for the 2 calls using inline assembly.

Note: Always remember to update the free memory pointer if the offset it points to is already used, to avoid solidity overriding the data stored there or using the value stored there in an unexpected way.

Also note to avoid overwriting the zero slot (0x60 memory offset) if you have undefined dynamic memory values within that call stack. An alternative is to explicitly define dynamic memory values or if used, to set the slot back to 0x00 before exiting the assembly block.

9. Use assembly to reuse memory space when creating more than one contract.

Solidity treats contract creation similar to external calls that returns 32 bytes (i.e it returns the address of the created contract or address(0) if the contract creation failed).

From the section on saving gas with external calls, we can immediately see that one way we can optimize this is to store the returned address in the scratch space and avoid expanding memory.

See a similar example below;

```
contract Solidity {
    // cost: 261032
    function call() external returns (Called, Called) {
        Called called1 = new Called();
        Called called2 = new Called();
        return (called1, called2);
    }
}


contract Assembly {
    // cost: 260210
    function call() external returns(Called, Called) {
```

```
            bytes memory creationCode = type(Called).creationCode;
            assembly {
                let called1 := create(0x00, add(0x20, creationCode),
    mload(creationCode))
                let called2 := create(0x00, add(0x20, creationCode),
    mload(creationCode))

                // revert if either called1 or called2 returned address(0)
                if iszero(and(called1, called2)) {
                    revert(0x00, 0x00)
                }

                mstore(0x00, called1)
                mstore(0x20, called2)

                return(0x00, 0x40)
            }
        }
    }



    contract Called {
        function add(uint256 a, uint256 b) external pure returns(uint256) {
            return a + b;
        }
    }
```

We saved close to 1,000 gas by using inline assembly.

Note: In the scenario where the two contracts to be deployed are not the same, the second contract's creation code would need to be mstored manually using inline assembly and not assigned to a variable in solidity to avoid memory expansion.

10. Test if a number is even or odd by checking the last bit instead of using a modulo operator

The conventional way to check if a number is even or odd is to do x % 2 == 0 where x is the number in question. You can instead check if x & uint256(1) == 0. where x is assumed to be a uint256. Bitwise and is cheaper than the modulo op code. In binary, the rightmost bit represents "1" whereas all the bits to the are multiples of 2, which are even. Adding "1" to an even number causes it to be odd.

Solidity Compiler Related

The following tricks are known to improve gas efficiency in the Solidity compiler. However, it is expected that the Solidity compiler will improve over time making these tricks less useful or even counterproductive.

You shouldn't blindly use the tricks listed here, but benchmark both alternatives.

Some of these tricks are already incorporated by the compiler when using the --via-ir compiler flag, and may even make the code less efficient when that flag is used.

Benchmark. Always benchmark.

1. Prefer strict inequalities over non-strict inequalities, but test both alternatives

It is generally recommended to use strict inequalities (<, >) over non-strict inequalities (<=, >=). This is because the compiler will sometimes change a > b to be !(a < b) to accomplish the non-strict inequality. The EVM does not have an opcode for checking less-than-or-equal to or greater-than-or-equal to.

However, you should try both comparisons, because it is not always the case that using the strict inequality will save gas. This is very dependent on the context of the surrounding opcodes.

2. Split require statements that have boolean expressions

When we split require statements, we are essentially saying that each statement must be true for the function to continue executing.

If the first statement evaluates to false, the function will revert immediately and the following require statements will not be examined. This will save the gas cost rather than evaluating the next require statement.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Require {
    function dontSplitRequireStatement(uint256 x, uint256 y) external
pure returns (uint256) {
```

```
        require(x > 0 && y > 0); // both conditon would be evaluated,
before reverting or notreturn x * y;
    }
}

contract RequireTwo {
    function splitRequireStatement(uint256 x, uint256 y) external pure
returns (uint256) {
        require(x > 0); // if x <= 0, the call reverts and "y > 0" is
not checked.
        require(y > 0);

        return x * y;
    }
}
```

3. Split revert statements

Similar to splitting require statements, you will usually save some gas by not having a boolean operator in the if statement.

```
contract CustomErrorBoolLessEfficient {
    error BadValue();

    function requireGood(uint256 x) external pure {
        if (x < 10 || x > 20) {
            revert BadValue();
        }
    }
}

contract CustomErrorBoolEfficient {
    error TooLow();
    error TooHigh();

    function requireGood(uint256 x) external pure {
        if (x < 10) {
            revert TooLow();
        }
        if (x > 20) {
            revert TooHigh();
        }
```

```
        }
    }
```

4. Always use Named Returns

The solidity compiler outputs more efficient code when the variable is declared in the return statement. There seem to be very few exceptions to this in practice, so if you see an anonymous return, you should test it with a named return instead to determine which case is most efficient.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract NamedReturn {
    function myFunc1(uint256 x, uint256 y) external pure returns
(uint256) {
        require(x > 0);
        require(y > 0);

        return x * y;
    }
}

contract NamedReturn2 {
    function myFunc2(uint256 x, uint256 y) external pure returns
(uint256 z) {
        require(x > 0);
        require(y > 0);

        z = x * y;
    }
}
```

5. Invert if-else statements that have a negation

This is the same example we gave at the beginning of the article. In the code snippet below, the second function avoids an unnecessary negation. In theory, the extra ! increases the computational cost. But as we noted at the top of the article, you should benchmark both methods because the compiler is can sometimes optimize this.

```
function cond() public {
    if (!condition) {
        action1();
    }
```

```
        else {
            action2();
        }
    }

    function cond() public {
        if (condition) {
            action2();
        }
        else {
            action1();
        }
    }
```

6. Use ++i instead of i++ to increment

The reason behind this is in way ++i and i++ are evaluated by the compiler.
i++ returns i(its old value) before incrementing i to a new value. This means that 2 values are stored on the stack for usage whether you wish to use it or not. ++i on the other hand, evaluates the ++ operation on i (i.e it increments i) then returns i (its incremented value) which means that only one item needs to be stored on the stack.

7. Use unchecked math where appropriate

Solidity uses checked math (i.e it reverts if the result of a math operation overflows the type of the result variable) by default, but there are some situations where overflow is infeasible to occur.

- **for loops which have natural upper bounds**
- **math where the input to the function is already sanitized into reasonable ranges**
- **variables that start at a low number and then each transaction adds one or a small number to it (like a counter)**

Whenever you see arithmetic in code, ask yourself if there is a natural guard to overflow or underflow in the context (keep in mind the type of the variable holding the number too). If so, add an unchecked block.

8. Write gas-optimal for-loops

Note: As of Solidity 0.8.22, this trick is done automatically by the compiler and does not need to be done explicitly.

This is what a gas-optimal for loop looks like, if you combine the two tricks above:

```
    for (uint256 i; i < limit; ) {
```

```
        // inside the loop

        unchecked {
            ++i;
        }
    }
```

The two differences here from a conventional for loop is that i++ becomes ++i (as noted above), and it is unchecked because the limit variable ensures it won't overflow.

9. Do-While loops are cheaper than for loops

If you want to push optimization at the expense of creating slightly unconventional code, Solidity do-while loops are more gas efficient than for loops, even if you add an if-condition check for the case where the loop doesn't execute at all.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

// times == 10 in both tests
contract Loop1 {
    function loop(uint256 times) public pure {
        for (uint256 i; i < times;) {
            unchecked {
                ++i;
            }
        }
    }
}

contract Loop2 {
    function loop(uint256 times) public pure {
        if (times == 0) {
            return;
        }

        uint256 i;

        do {
            unchecked {
                ++i;
            }
```

```
            } while (i < times);
        }
    }
```

10. Avoid Unnecessary Variable Casting, variables smaller than uint256 (including boolean and address) are less efficient unless packed

It is better to use uint256 for integers, except when smaller integers are necessary.
This is because the EVM automatically converts smaller integers to uint256 when they are used. This conversion process adds extra gas cost, so it is more efficient to use uint256 from the start.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Unnecessary_Typecasting {
    uint8 public num;

    function incrementNum() public {
        num += 1;
    }
}

// Uses less gas
contract NoTypecasting {
    uint256 public num;

    function incrementNumCheap() public {
        num += 1;
    }
}
```

11. Short-circuit booleans

In Solidity, when you evaluate a boolean expression (e.g the || (logical or) or && (logical and) operators), in the case of || the second expression will only be evaluated if the first expression evaluates to false and in the case of && the second expression will only be evaluated if the first expression evaluates to true. This is called short-circuiting.

For example, the expression require(msg.sender == owner || msg.sender == manager) will pass if the first expression msg.sender == owner evaluates to true. The second expression msg.sender == manager will not be evaluated at all.

However, if the first expression msg.sender == owner evaluates to false, the second expression msg.sender == manager will be evaluated to determine whether the overall expression is true or false. Here, by checking the condition that is most likely to pass firstly, we can avoid checking the second condition thereby saving gas in majority of successful calls.

This is similar for the expression require(msg.sender == owner && msg.sender == manager). If the first expression msg.sender == owner evaluates to false, the second expression msg.sender == manager will not be evaluated because the overall expression cannot be true. For the overall statement to be true, both side of the expression must evaluate to true. Here, by checking the condition that is most likely to fail firstly, we can avoid checking the second condition thereby saving gas in majority of call reverts.

Short-circuiting is useful and it's recommended to place the less expensive expression first, as the more costly one might be bypassed. If the second expression is more important than the first, it might be worth reversing their order so that the cheaper one gets evaluated first.

12. Don't make variables public unless it is necessary to do so

A public storage variable has an implicit public function of the same name. A public function increases the size of the jump table and adds bytecode to read the variable in question. That makes the contract larger.
Remember, private variables aren't private, it's not difficult to extract the variable value using web3.js. This is especially true for constants which are meant to be read by humans rather than smart contracts.

13. Prefer very large values for the optimizer

The Solidity optimizer focuses on optimizing two primary aspects:

1. The deployment cost of a smart contract.
2. The execution cost of functions within the smart contract.

There's a trade-off involved in selecting the runs parameter for the optimizer.
Smaller run values prioritize minimizing the deployment cost, resulting in smaller creation code but potentially unoptimized runtime code. While this reduces gas costs during deployment, it may not be as efficient during execution.
Conversely, larger values of the runs parameter prioritize the execution cost. This leads to larger creation code but an optimized runtime code that is more cheaper to execute. While this may not significantly affect deployment gas costs, it can significantly reduce gas costs during execution.
Considering this trade-off, if your contract will be used frequently it is advisable to use a larger value for the optimizer. As this will save up gas costs in a long term.

14. Heavily used functions should have optimal names

The EVM uses a jump table for function calls, and function selectors with lesser hexadecimal order are sorted first over selectors with higher hex order. In other words, if two function selectors, for example, 0x000071c3 and 0xa0712d68, are present in the same contract, the function with the selector 0x000071c3 will be checked before the one with 0xa0712d68 during contract execution.
Hence, if a function is used frequently, it is essential for it to have an optimal name. This optimization increases its chances of being sorted first, thus saving gas costs from further checks (although if there are more than four functions in the contract, the EVM does a binary search for the jump table instead of a linear search).
This also reduces calldata cost (if the function has leading zeros, as zero bytes cost 4 gas, and non-zero bytes cost 16 gas).

Here is a good demo below.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract FunctionWithLeadingZeros {
    uint256 public totalSupply;

    // selector = 0xa0712d68
    function mint(uint256 amount) public {
        totalSupply += amount;
    }

    // selector = 0x000071c3 (this cheaper than the above function)
    function mint_184E17(uint256 amount) public {
        totalSupply += amount;
    }
}
```

In addition, we have a helpful tool called Solidity Zero Finder which was built with Rust that can assist developers in achieving this. It is available at this GitHub repository.

15. Bitshifting is cheaper than multiplying or dividing by a power of two

In Solidity, it is often more gas efficient to multiply or divide numbers that are powers of two by shifting their bits, rather than using the multiplication or division operators.
For example, the following two expressions are equivalent

```
10 * 2
10 << 1 # shift 10 left by 1
```

and this is also equivalent

```
8 / 4
8 >> 2 # shift 8 right by 2
```

Bit shifting operations opcodes in the EVM, such as shr (shift right) and shl (shift left), cost 3 gas while multiplication and division operations (mul and div) cost 5 gas each.

Majority of gas savings also comes the fact that solidity does no overflow/underflow or division by check for shr and shl operations. It's important to have this in mind when using these operators so that overflow and underflow bugs don't happen.

16. It is sometimes cheaper to cache calldata

Although the calldataload instruction is a cheap opcode, the solidity compiler will sometimes output cheaper code if you cache calldataload. This will not always be the case, so you should test both possibilities.

```
contract LoopSum {
    function sumArr(uint256[] calldata arr) public pure returns
(uint256 sum) {
        uint256 len = arr.length;
        for (uint256 i = 0; i < len; ) {
            sum += arr[i];
            unchecked {
                ++i;
            }
        }
    }
}
```

17. Use branchless algorithms as a replacement for conditionals and loops

The max code from an earlier section is an example of a branchless algorithm, i.e. it eliminates the JUMP opcode, which is more costly than arithmetic opcodes in general.

For loops have jumps built into them, so you may want to consider loop unrolling to save gas.

Loops don't have to be unrolled all the way. For example, you can execute a loop two items at a time and cut the number of jumps in half.

This is a very extreme optimization, but you should be aware that conditional jumps and loops introduce a slightly more expensive opcode.

18. Internal functions only used once can be inlined to save gas

It is okay to have internal functions, however they introduce additional jump labels to the bytecode.

Hence, in a case where it is only used by one function, it is better to inline the logic of the internal function inside the function it is being used. This will save some gas by avoiding jumps during the function execution.

19. Compare array equality and string equality by hashing them if they are longer than 32 bytes

This is a trick you will rarely use, but looping over the arrays or strings is a lot costlier than hashing them and comparing the hashes.

20. Use lookup tables when computing powers and logarithms

If you need to take logarithms or powers where the base or the power is a fraction, it may be preferable to precompute a table if either the base or the power is fixed.
Consider the Bancor Formula and Uniswap V3 Tick Math as examples.

21. Precompiled contracts may be useful for some multiplication or memory operations.

The Ethereum precomipled contracts provide operations primarily for cryptography, but if you need to multiply large numbers over a modulus or copy significant chunk of memory, consider using the precompiles. Be aware that this might make your application incompatible with some layer 2s.

22. n * n * n may be cheaper than n ** 3

Two MUL op codes is 10 gas total, but the EXP op code costs 10 gas + 50 * (exponent size in bytes).

Dangerous techniques

If you are participating in a gas optimization contest, then these unusual design patterns can help, but using them in production is highly discouraged, or at the very least should be done with extreme caution.

1. Use gasprice() or msg.value to pass information

Passing parameters to a function will at bare minimum add 128 gas, because each zero byte of calldata costs 4 gas. However, you can set the gasprice or msg.value for free to pass numbers that way. This of course won't work in production because msg.value costs real Ethereum and if your gas price is too low, the transaction won't go through, or will waste cryptocurrency.

2. Manipulate environment variables like coinbase() or block.number if the tests allow it

This of course will not work in production, but it can serve as a side channel to modify a smart contract's behavior.

3. Use gasleft() to branch decisions at key points

Gas is used up as the execution progresses, so if you want to do something like terminate a loop after a certain point or change behavior in a later part of the execution, you can use the gasprice() functionality to branch decision making. gasleft() decrements for "free" so this saves gas.

4. Use send() to move ether, but don't check for success

The difference between send and transfer is that transfer reverts if the transfer fails, but send returns false. However, you can just ignore the return value of send, and that will result in fewer op codes. Ignoring return values is a very bad practice, and it's a shame the compiler doesn't stop you from doing that. In production systems, you should not use send() at all because of the gas limit.

5. Make all functions payable

This is a controversial optimization because it allows for an unexpected state change in a transaction, and doesn't save that much gas. But in the context of a gas contest, make all functions payable to avoid the extra opcodes that check if msg.value is nonzero.

As noted earlier, setting the constructor or admin functions to be payable is a legitimate way to save gas, since presumably the deployer and admin know what they are doing and can do more destructive things than send ether.

6. External library jumping

Solidity traditionally uses 4 bytes and a jump table to determine which function to use. However, one can (very unsafely!) simply supply the jump destination as a calldata argument, reducing the "function selector" down to one byte and avoiding the jump table altogether. More information can be seen in this tweet.

7. Append bytecode to the end of the contract to create a highly optimized subroutine

Some computationally intensive algorithms, such as hash functions, are better written in raw bytecode rather than Solidity, or even Yul. For example, Tornado Cash writes the MiMC hash function as a separate smart contract, written directly in raw bytecode. Avoid the extra 2,600 or 100 gas cost (cold or warm access) of another smart contract by appending that bytecode to the actual contract and jumping back and forth from it. Here is a proof of concept using Huff.

Outdated tricks

1. external is cheaper than public

You should still prefer the external modifier for clarity sake if the function cannot be called inside the contract, but it has no effect on gas savings.

2. != 0 is cheaper than > 0

Somewhere around solidity 0.8.12 or so, this stopped being true. If you are forced to use an old version, you can still benchmark it.

Bad Practices

There are several common mistakes developers make that lead to higher gas costs. Due to space limitations, we've published this list in a separate article.

Learn more with RareSkills

Learning is always more effective when surrounded by a motivated community and guided by seasoned instructors. This material is part of our advanced solidity bootcamp. If you want to practice gas optimization with other solidity professionals under the guidance of industry leaders, check out the program!