**Introduction to Storage optimization**

**Optimizing GAS consumption by properly managing Storage**

**Storage** is the specific area within the blockchain where data associated with a smart contract is permanently saved. These are the variables that we defined at the top of our contract, before going into functions, also called **state variables** or **global variables**.

Imagine yourself being in a giant locker room, and in each locker, you have a space of 32 bytes. Each locker (storage space) is numbered/labeled and its number/label acts as the key, in the key-value pair, thus using this number/label we can access what's stored in the locker. Think of state variables as the labels you give to these lockers, allowing you to easily retrieve the information you've stored. But remember, space on this shelf isn't unlimited, and every time you add or remove something, it comes at a computational cost. From the previous lessons, we learned that this computational cost bears the name of gas.

So, being a tidy developer, you'll want to use your storage efficiently, potentially packing smaller data types together or using mappings (fancy ways of labeling sections of the locker) to keep things orderly and cost-effective.

The following info is a bit more advanced, but please take your time and learn it. As we emphasized in the previous lesson, the amount of gas people pay for in interacting with your protocol can be an important element for their retention, regardless of the type of web3 protocol you'll build.

***No one likes paying huge transaction costs.***

**Layout of State Variables in Storage**

The [Solidity documentation](https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html) is extremely good for understanding this subject.

The important aspects are the following:

* Each storage has 32 bytes;
* The slots numbering starts from 0;
* Data is stored contiguously starting with the first variable placed in the first slot;
* Dynamically-sized arrays and mappings are treated differently (we'll discuss them below);
* The size of each variable, in bytes, is given by its type;
* If possible, multiple variables < 32 bytes are packed together;
* If not possible, a new slot will be started;
* Immutable and Constant variables are baked right into the bytecode of the contract, thus they don't use storage slots.

Now this seems like a lot, but let's go through some examples: (Try to think about how the storage looks before reading the description)

uint256 var1 = 1337;

uint256 var2 = 9000;

uint64 var3 = 0;

How are these stored?

In slot 0 we have var1, in slot 1 we have var2, and in slot 3 we have var 3. Because var 3 only used 8 bytes, we have 24 bytes left in that slot. Let's try another one:

uint64 var1 = 1337;

uint128 var2 = 9000;

bool var3 = true;

bool var4 = false;

uint64 var5 = 10000;

address user1 = 0x1F98431c8aD98523631AE4a59f267346ea31F984;

uint128 var6 = 9999;

uint8 var7 = 3;

uint128 var8 = 20000000;

How are these stored?

Let's structure them better this time:

slot 0

* var1 8 bytes (8 total)
* var2 16 bytes (24 total)
* var3 1 byte (25 total)
* var4 1 byte (26 total)
* var5 has 8 bytes, it would generate a total of 34 bytes, but we have only 32 so we start the next slot

slot 1

* var5 8 bytes (8 total)
* user1 20 bytes (28 total)
* var6 has 8 bytes, it would generate a total of 36 bytes, we have a max of 32 so we start the next slot

slot2

* var6 16 byes (16 total)
* var7 1 byte (17 total)
* var8 has 16 bytes, it would generate a total of 33 bytes, but as always we have only 32, we start the next slot

slot3

* var8 16 bytes (16 total)

Can you spot the inefficiency? slot 0 has 6 empty bytes, slot 1 has 4 empty bytes, slot 2 has 15 empty bytes, slot 3 has 16 empty bytes. Can you come up with a way to minimize the number of slots?

What would happen if we move var7 between var4 and var5, so we fit its 1 byte into slot 0, thus reducing the total of slot2 to 16 bytes, leaving enough room for var8 to fit in. You get the gist.

The total bytes of storage is 87. We divide that by 32 and we find out that we need at least 2.71 slots ... which means 3 slots. We cannot reduce the number of slots any further.

Mappings and Dynamic Arrays can't be stored in between the state variables as we did above. That's because we don't quite know how many elements they would have. Without knowing that we can't mitigate the risk of overwriting another storage variable. The elements of mappings and dynamic arrays are stored in a different place that's computed using the Keccak-256 hash. Please read more about this [here](https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html#mappings-and-dynamic-arrays).

**Back to FundMe**

Make sure that you have the following getter in FundMe.sol:

function getPriceFeed() public view returns (AggregatorV3Interface) {

return s\_priceFeed;

}

Please add the following function in your `FundMe.t.sol`:

function testPrintStorageData() public {

for (uint256 i = 0; i < 3; i++) {

bytes32 value = vm.load(address(fundMe), bytes32(i));

console.log("Value at location", i, ":");

console.logBytes32(value);

}

console.log("PriceFeed address:", address(fundMe.getPriceFeed()));

}

In the test above we used a new cheatcode: vm.load. Its sole purpose is to load the value found in the provided storage slot of the provided address. Read more about it [here](https://book.getfoundry.sh/cheatcodes/load).

Run the test above by calling this in your terminal:

forge test --mt testPrintStorageData -vv

Ran 1 test for test/FundMe.t.sol:FundMeTest

[PASS] testPrintStorageData() (gas: 19138)

Logs:

Value at location 0 :

0x0000000000000000000000000000000000000000000000000000000000000000

Value at location 1 :

0x0000000000000000000000000000000000000000000000000000000000000000

Value at location 2 :

0x00000000000000000000000090193c961a926261b756d1e5bb255e67ff9498a1

PriceFeed address: 0x90193C961A926261B756D1E5bb255e67ff9498A1

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 771.50µs (141.90µs CPU time)

Let's interpret the data above:

* In slot 0 we have a bytes32(0) stored (or 32 zeroes). This happened because the first slot is assigned to the s\_addressToAmountFunded mapping.
* In slot 1 we have a bytes32(0) stored. This happened because the second slot is assigned to the s\_funders dynamic array.
* In slot 2 we have 0x00000000000000000000000090193c961a926261b756d1e5bb255e67ff9498a1 stored. This is composed of 12 pairs of zeroes (12 x 00) corresponding to the first 12 bytes and 90193c961a926261b756d1e5bb255e67ff9498a1. If you look on the next line you will see that is the priceFeed address.

This is one of the methods of checking the storage of a smart contract.

Another popular method is using forge inspect. This is used to obtain information about a smart contract. Read more about it [here](https://book.getfoundry.sh/reference/forge/forge-inspect).

Call the following command in your terminal:

forge inspect FundMe storageLayout

If we scroll to the top we will find a section called storage. Here you can find the label, the type and the slot it corresponds to. It's simpler than using vm.load but vm.load is more versatile, as in you can run tests against what you expect to be stored vs what is stored.

Another method of checking a smart contract's storage is by using cast storage. For this one, we need a bit of setup.

Open a new terminal, and type anvil to start a new anvil instance.

Deploy the fundMe contract using the following script:

forge script DeployFundMe --rpc-url http://127.0.0.1:8545 --private-key 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 --broadcast

The rpc url used is the standard anvil rpc. The private key used is the first of the 10 private keys anvil provides.

In the Return section just printed in the terminal, we find the address of the newly deployed fundMe. In my case, this is 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512.

Call the following command in your terminal:

cast storage 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512 2

This prints what's stored in slot number 2 of the fundMe contract. This checks with our previous methods of finding what's in slot number 2 (even if the address is different between methods).

**Very important note: the word private has multiple meanings in the** [**Merriam Webster dictionary**](https://www.merriam-webster.com/dictionary/private)**. Always remember that the keyword private attached to a variable/function means that the variable/function has restricted access from other parties apart from the main contract.**

**THIS DOESN'T MEAN THE VARIABLE IS A SECRET!!!**

**Everything on the blockchain is public. Any smart contract's storage can be accessed in one of the 3 ways we showed in this lesson. Do not share sensitive information or store passwords inside smart contracts, we can all read them.**

Storage is one of the harder Solidity subjects. Mastering it is one of the key prerequisites in writing gas-efficient and tidy smart contracts.

Congratz for getting to this point! Up next we will optimize the withdraw function.