**Optimise the withdraw function gas costs**

**Making the withdraw function more gas-efficient**

In the previous lesson, we talked about storage. But why is storage management important? Simple, reading and writing from storage is a very expensive operation.

Let's explore this subject more.

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`

Copy the `fundMe` contract address.

Run the following command:

(replace the address here with the address of your newly deployed `fundMe` contract)

cast code 0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9`

Ok, the output looks like an extremely big chunk of random numbers and letters. Perfect!

Something to the extent of 0x608060405260043610610...736f6c63430008130033. Copy the entire thing and put it in [here](https://etherscan.io/opcode-tool). Thus we obtain the Decoded Bytecode which is a list of Opcodes.

[1] PUSH1 0x80

[3] PUSH1 0x40

[4] MSTORE

[6] PUSH1 0x04

[7] CALLDATASIZE

[8] LT

[11] PUSH2 0x008a

[12] JUMPI

[14] PUSH1 0x00

[15] CALLDATALOAD

[17] PUSH1 0xe0

[18] SHR

[19] DUP1

[24] PUSH4 0x893d20e8

[25] GT

[28] PUSH2 0x0059

[...]

These look readable! But what are we reading?

Opcodes (short for operation codes) are the fundamental instructions that the EVM understands and executes. These opcodes are essentially the building blocks that power smart contract functionality. You can read about each opcode [here](https://www.evm.codes/).

In that table alongside the description, you will find the bytecode number of each opcode, the name of the opcode, the minimum gas it consumes and the input/output. Please be mindful of the gas each opcode costs. Scroll down the list until you get to the 51-55 opcode range.

As you can see an MLOAD/MSTORE has a minimum gas cost of 3 and a SLOAD/SSTORE has a minimum gas of 100 ... that's over 33x. And keep in mind these are minimums. The difference is usually bigger. This is why we need to be careful with saving variables in storage, every time we access or modify them we will be forced to pay way more gas.

Let's take a closer look at the withdraw function.

We start with a for loop, that is initializing a variable called funderIndex in memory and compares it with s\_funders.length on every loop iteration. As you know s\_funders is the private array that holds all the funder's addresses, currently located in the state variables zone. If we have 1000 funders, we will end up reading the length of the s\_funders array 1000 times, paying the SLOAD costs 1000 times. This is extremely inefficient.

Let's rewrite the function. Add the following to your FundMe.sol:

function cheaperWithdraw() public onlyOwner {

uint256 fundersLength = s\_funders.length;

for(uint256 funderIndex = 0; funderIndex < fundersLength; funderIndex++) {

address funder = s\_funders[funderIndex];

s\_addressToAmountFunded[funder] = 0;

}

s\_funders = new address[](0);

(bool callSuccess,) = payable(msg.sender).call{value: address(this).balance}("");

require(callSuccess, "Call failed");

}

First, let's cache the s\_funders length. This means we create a new variable, inside the function (to be read as in memory) so if we read it 1000 times we don't end up paying a ridiculous amount of gas.

Then let's integrate this into the for loop.

The next step is getting the funder's address from storage. Sadly we can't avoid this one. After this we zero the recorded amount in the s\_addressToAmountFunded mapping, also we can't avoid this. We then reset the s\_funders array, and send the ETH. Both these operations cannot be avoided.

Let's find out how much we saved. Open FundMe.t.sol.

Let's copy the testWithdrawFromMultipleFunders function and replace the withdraw function with cheaperWithdraw.

function testWithdrawFromMultipleFundersCheaper() public funded {

uint160 numberOfFunders = 10;

uint160 startingFunderIndex = 1;

for (uint160 i = startingFunderIndex; i < numberOfFunders + startingFunderIndex; i++) {

// we get hoax from stdcheats

// prank + deal

hoax(address(i), SEND\_VALUE);

fundMe.fund{value: SEND\_VALUE}();

}

uint256 startingFundMeBalance = address(fundMe).balance;

uint256 startingOwnerBalance = fundMe.getOwner().balance;

vm.startPrank(fundMe.getOwner());

fundMe.cheaperWithdraw();

vm.stopPrank();

assert(address(fundMe).balance == 0);

assert(startingFundMeBalance + startingOwnerBalance == fundMe.getOwner().balance);

assert((numberOfFunders + 1) \* SEND\_VALUE == fundMe.getOwner().balance - startingOwnerBalance);

}

Now let's call forge snapshot. If we open .gas-snapshot we will find the following at the end:

FundMeTest:testWithdrawFromMultipleFunders() (gas: 535148)

FundMeTest:testWithdrawFromMultipleFundersCheaper() (gas: 534219)

As you can see, we saved up 929 gas just by caching one variable.

One of the reasons we easily identified this optimization was the use of s\_ in the s\_funders array declaration. The ability to know, at any time, what comes from storage and what is in memory facilitates this type of optimization. That's why we recommend using the s\_ and i\_ and all upper case for constants, to always know what comes from where. Familiarize yourself with the style guide available [here](https://docs.soliditylang.org/en/v0.8.4/style-guide.html).