9️⃣

Basics IX: Gas Optimisation

Avoiding gas fees completely is impossible. However, reducing gas fees by optimising the code can result in dramatically more profitable projects since gas fees add up by time. From this point of view, even pretty simple gas optimisation in Solidity can make a huge difference, at least over time.

What is cheap:

1. Reading constants and immutable variables.
2. Reading and writing local variables.
3. Reading and writing memory variables like memory arrays and structs.
4. Reading calldata variables like calldata arrays and structs.
5. Internal function calls.

What is expensive:

1. Read and writing state variables that are stored in contract storage.
2. External function calls.
3. Loops

`for` Loop & `SafeMath` Optimisation

```
contract Gas_Test{

    // Definition of state variables
    uint[] public arrayFunds;
    uint public totalFunds;

    // A constructor to populate the “arrayFunds” variable
    constructor() {
        arrayFunds = [1,2,3,4,5,6,7,8,9,10,11,12,13];
    }

    function unsafe_inc(uint x) private pure returns (uint) {
        unchecked { return x + 1; }
    }

    function optionA() external {
```

```
        for (uint i =0; i < arrayFunds.length; i++){
            totalFunds = totalFunds + arrayFunds[i];
        }
    }

    function optionB() external {
        uint _totalFunds;
        for (uint i =0; i < arrayFunds.length; i++){
            _totalFunds = _totalFunds + arrayFunds[i];
        }
        totalFunds = _totalFunds;
    }

    function optionC() external {
        uint _totalFunds;
        uint[] memory _arrayFunds = arrayFunds;
        for (uint i =0; i < _arrayFunds.length; i++){
            _totalFunds = _totalFunds + _arrayFunds[i];
        }
        totalFunds = _totalFunds;
    }

    function optionD() external {
        uint _totalFunds;
        uint[] memory _arrayFunds = arrayFunds;
        for (uint i =0; i < _arrayFunds.length; i = unsafe_inc(i)){
            _totalFunds = _totalFunds + _arrayFunds[i];
        }
        totalFunds = _totalFunds;
    }

}
```

optionA is spending a lot of fees as it reads from a state variable and writes to a state variable in each iteration of the for loop. **optionA** is reading and writing directly to the blockchain in each iteration of the loop. We can optimise that function by caching our variable to a memory variable, then we can use the memory variable inside the loop to reduce gas fee. Check **optionB** for this solution.

Just by this simple tweak in **optionB**, we can save a lot on fees during the execution of our “*for*”
loop. However, even though we are not writing to the blockchain in the loop, we are still reading from the blockchain for each iteration. Thus, this is a clear indication that we can take gas optimizations in Solidity even further. This takes us to the **optionC** function.

By adding another memory variable, we are now caching both of our state variables `arrayFunds` and `totalFunds` to memory variables `_arrayFunds` and `_totalFunds`. With the above memory array, we are also not reading from the blockchain for each of the loop’s iterations. As such, we only read from the blockchain once before initializing

the loop. Then, we execute our function with the copy of the array, which is in memory. Finally, we just populate our variable as we did in “optionB”.

> By taking both the reading and the writing within the “*for*” loop iterations off the chain, we made quite a difference. However, we wanted to take things even further.

`SafeMath` and Unchecked Arithmetic

In the past, Solidity didn’t revert to variable overflow. Whenever you tried to store more or store a figure that was greater than that upper limit in Solidity’s previous version, it didn’t return an error. Instead, it gave an incorrect value. Because of this, `SafeMath` was developed to solve that issue. However, **starting from the Solidity version 0.8**, this flaw was fixed. As such, Solidity was able to revert on overflows, which also eliminated the need for `SafeMath`. *Though, this made the arithmetic more expensive in terms of gas*.

Let’s consider `i++` used in all of our function variations in the code presented above. In this kind of addition of the variable `i`, we use the protected arithmetic of Solidity, also known as “checked arithmetic”.

As far as the gas fees go, it would be cheaper to use unchecked arithmetic. Fortunately, we can do this confidently because it would be quite difficult for our variable `i` to overflow. The latter is the `uint256` variable, which has a pretty high limit. Moreover, we know that no array will be as long as that limit.

As part of this advanced gas optimization in Solidity, we’ll add a helper function.

```
function unsafe_inc(uint x) private pure returns (uint) {
        unchecked { return x + 1; }
}
```

Create a new function called `optionD` to use unchecked arithmetic.

```
function optionD() external {
        uint _totalFunds;
        uint[] memory _arrayFunds = arrayFunds;
        for (uint i =0; i < _arrayFunds.length; i = unsafe_inc(i)){
            _totalFunds = _totalFunds + _arrayFunds[i];
        }
        totalFunds = _totalFunds;
}
```

Consequently,

- **optionA** – Includes on-chain reading and writing throughout all of the loop's iterations.
- **optionB** – Includes on-chain reading throughout all of the loop's iterations but off-chain writing.
- **optionC** – Includes off-chain reading and writing throughout all of the loop's iterations.
- **optionD** – Includes off-chain reading and writing throughout all of the loop's iterations in combination with the unchecked arithmetic.

Comparison of gas fees after the optimisations have been done:

optionD < **optionC** < **optionB** < **optionA**

Packing Structs

When values are read or written in contract storage a full 256 bits are read or written. So if you can pack multiple variables within one 256 bit storage slot then you are cutting the cost to read or write those storage variables in half or more.

Unoptimised:

```
struct MyStruct {
  uint256 myTime;
  address myAddress;
}
```

Optimised:

```
struct MyStruct {
  uint96 myTime;
  address myAddress;
}
```

Data Types

It is better to use `uint256` and `bytes32` than using `uint8` for example. While it seems like `uint8` will consume less gas than `uint256` it is not true, since the Ethereum virtual Machine(EVM) will still occupy 256 bits, fill 8 bits with the `uint` variable and fill the extra bites with zeros.