Arrays in Solidity

Filtering an Array

Copy page

Explore techniques to filter an array.

In this exercise, you'll explore two different solutions for filtering an array in Solidity. By doing so, you'll gain a better understanding of the constraints present while working with arrays, and have the chance to learn and compare the gas costs of different approaches.

Objectives

By the end of this lesson you should be able to:

- Write a function that can return a filtered subset of an array

First Pass Solution

Setup

Create a new workspace in Remix and add a file called `ArrayDemo.sol` containing a `contract` called `ArrayDemo`. Initialize an array containing the numbers from 1 to 10. Add a stub for a function called `getEvenNumbers` that returns a `uint[] memory`.

```
contract ArrayDemo {
    uint[] public numbers = [1,2,3,4,5,6,7,8,9,10];

    function getEvenNumbers() external view returns(uint[] memory) {
        // TODO
    }
}
```

⚠ You don't have to declare the size of the memory array to be returned. You usually don't want to either, unless the results will always be the same, known size.

Finding the Number of Even Numbers

We need to initialize a `memory` array to hold the results, but to do so, we need to know how big to make the array. Don't be tempted to count the number of evens in `numbers`, as what happens if we modify it later?

The simple and obvious solution is to simply iterate through `numbers` and count how many even numbers are present. You could add that functionality in `getEvenNumbers()`, but it might be useful elsewhere, so a better practice would be to separate these concerns into another function.

Go ahead and write it on your own. It needs to:

- Instantiate a `uint` to hold the results
- Iterate through all values in `numbers` and increment that number if the value is even
- Return the result

You should end up with something like:

Reveal code

The `_` in front of the function name is a practice used by some developers, in Solidity and in other languages, to indicate visually that this function is intended for internal use only.

Returning Only Even Numbers

Now that we have a method to find out how big the return array needs to be in `getEvenNumbers()`, we can simply loop through `numbers`, and add the even numbers to the array to be returned.

Finish the function on your own. It needs to:

- Determine the number of results and instantiate an array that size
- Loop through the `numbers` array and if a given number is even, add it to the next unused index in the results array

You should end up with something like:

Reveal code

Did you catch the compiler warning about `view`? You aren't modifying state, so you should mark it as such.

Testing the Function

Deploy your contract and test the function. You should get a return of `[2,4,6,8,10]`. The total gas cost will be about 63,947, depending on if you used the same helper variables, etc.

Optimizing the Function

It does seem inefficient to loop through the same array twice. What if we instead kept track of how many even numbers to expect. That way, we would only need to loop

once, thus saving gas! Right?

Only one way to find out.

Tracking Relevant Data

Add a contract-level variable called `numEven`, and initialize it with **5**, the number of even numbers in the array. Modify `getEvenNumbers()` to use `numEven` instead of the `_countEvenNumbers()` function. It should now look like:

Reveal code

Redeploy and test again. Success, the function now only costs about 57,484 gas to run! Except there is a catch. Remember, it's going to cost about 5000 gas to update `numEven` **each time** the array adds an even number.

A More Realistic Accounting

As we considered above, in a real-world example, we wouldn't declare the array up front, it would be modified over time. A slightly more realistic example would be to fill the array with a function.

Change the declaration for `numbers` and `numEven` so that they have their respective default values to begin with.

```
uint[] public numbers;
uint numEven;
```

Add a new function called `debugLoadArray` that takes a `uint` called `_number` as an argument, and fills the array by looping through `_number` times, pushing each number into the array. **For now, *don't* update** `numEven` .

Reveal code

Test out the function by loading in **10** numbers. It costs about 249,610 gas to load the array. Now, add functionality to **also** increment `numEven` when the number added is even. We can't just calculate it, because although the numbers are sequential in the debug function, they might not be in real world use.

Reveal code

Be sure to redeploy and try again with **10** numbers. This time, the cost was about 275,335 gas. That's almost 26,000 more gas in an effort to save the 5,000 gas needed to run `_countEvenNumbers()` .

Looking at the Big Picture

What about more? What if there are a thousand numbers in the array? What about a million?

Let's start with 500, any more will break the Remix EVM simulation, and/or would trigger an out of gas error because we're approaching the gas limit for the entire block.

Comment out the `if` statement in `debugLoadArray` that checks for even numbers and load 500 numbers. The Remix EVM should be able to handle this, but it might hang up for a moment, or even crash. (You can also do this experiment with 250 numbers instead.)

```
function debugLoadArray(uint _number) external {
    for(uint i = 0; i < _number; i++) {
        numbers.push(i);
        // if(i % 2 == 0) {
        //    numEven++;
        //}
    }
}
```

You'll get a result of about 11,323,132 gas to load the array. That's a lot! The target total gas for a single block is 15 million, and the limit is 30 million.

Try again with the code to increment `numEven`. You should get about 11,536,282, or an increase of about 213,150 gas.

Now, test out `getEvenNumbers()` using `numEven` vs. using `_countEvenNumbers()`. With `numEven`, it should cost about 1,578,741 gas to find the even numbers. Using

As is often the case with code, it depends. You might think that the experiment makes things obvious. Paying 213k gas up front to track `_numEven` results in a savings of over 400k gas when filtering for even numbers. Even better, you might realize that the upfront cost difference will be spread across all of your users over time, making them almost trivial. You also might think that it's possible that the filter function could be called dozens of times for each time 500 numbers are loaded.

These are all valid considerations that you should evaluate as you are developing your code solution to a business problem. One last critical element to consider is that there is only a gas cost to read from the blockchain if it's another contract calling the function. It **doesn't** cost any gas to call `view` or `pure` functions from a front end or app.

If `getEvenNumbers` will never be called by another contract, then using `numEven` might cost more for no benefit!

Conclusion

In this lesson, you've explored a few different approaches to a problem. You've learned how to filter an array, but more importantly, you've learned some of the specific considerations in blockchain development. Finally, you've seen that pushing 500 integers to an array, usually a trivial operation, is very large and very expensive on the EVM.

Arrays Guide
Fixed Size Arrays