Arrays in Solidity

Arrays

Copy page

An overview of how arrays work in Solidity.

Solidity arrays are collections of the same type, accessed via an index, the same as any other language. Unlike other languages, there are three types of arrays - *storage*, *memory*, and *calldata*. Each has their own properties and constraints.

Objectives

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

- Describe the difference between storage, memory, and calldata arrays

Storage, Memory, and Calldata

The `storage`, `memory`, or `calldata` keywords are required when declaring a new **reference type** variable. This keyword determines the **data location** where the variable is stored and how long it will persist.

Storage

The `storage` keyword is used to assign state variables that become a part of the blockchain as a part of the *storage* for your contract. These remain as assigned until modified, for the lifetime of the contract.

Storage is **very expensive** compared to most other environments. It costs a minimum of 20000 gas to **store a value** in a new storage slot, though it's cheaper to update that value after the initial assignment (~5000+ gas).

This cost isn't a reason to be afraid of using storage. In the long run, writing clear, maintainable, and logical code will always cost less than jumping through hoops to save gas here and there. Just be as thoughtful with storage on the EVM as you would be with computation in most other environments.

Memory

The `memory` keyword creates temporary variables that only exist within the scope in which they are created. Memory is less expensive than storage, although this is relative. There are often circumstances where it is cheaper to work directly in storage

rather than convert to memory and back. Copying from one location to another can be quite expensive!

Calldata

The `calldata` storage location is where function arguments are stored. It is non-modifiable and the Solidity docs recommend using it where possible to avoid unnecessary copying, because it can't be modified. You'll learn more about this later, but doing so can help prevent confusing bugs when calling a function from another contract that takes in values from that contract's `storage` .

Array Data Locations

Arrays behave differently based on their data location. Assignment behavior also depends on data location. To **quote the docs**:

> Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
>
> Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.

> Assignments from `storage` to a **local** storage variable also only assign a reference.
>
> All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

Storage Arrays

Arrays in `storage` are passed by *reference*. In other words, if you assign a *storage array* half a dozen names, any changes you make will always modify the original, underlying storage array.

```
contract StorageArray {
    // Variables declared at the class level are always `storage`
    uint[] arr = [1, 2, 3, 4, 5];

    function function_1() public {
        uint[] storage arr2 = arr;

        arr2[0] = 99; // <- arr is now [99, 2, 3, 4, 5];
    }
}
```

You cannot use a `storage` array as a function parameter, and you cannot write a function that `return` s a `storage` array.

Storage arrays are dynamic, unless they are declared with an explicit size. However, their functionality is limited compared to other languages. The `.push(value)` function works as expected. The `.pop()` function removes the last value of an array, but it **does not** return that value. You also **may not** use `.pop()` with an index to remove an element from the middle of an array, or to remove more than one element.

You can use the `delete` keyword with an array. Doing so on an entire array will reset the array to zero length. Calling it on an element within the array will reset that value to its default. It **will not** resize the array!

```
uint[] arr_2 = [1, 2, 3, 4, 5];
function function_2(uint _num) public returns(uint[] memory) {
    arr_2.push(_num); // <- arr_2 is [1, 2, 3, 4, 5, <_num>]

    delete arr_2[2];  // <- arr_2 is [1, 2, 0, 4, 5, <_num>]

    arr_2.pop();  // <- arr_2 is [1, 2, 0, 4, 5] (_num IS NOT returned by .

    delete arr_2; // <- arr_2 is []

    return arr_2; // <- returns []
}
```

Storage arrays are implicitly convertible to `memory` arrays.

Memory Arrays

Arrays declared as `memory` are temporary and only exist within the scope in which they are created. Arrays in `memory` are **not** dynamic and must be declared with a fixed size. This can be done at compile time, by declaring a size inside the `[]` or during runtime by using the `new` keyword. Finally, `memory` arrays can be implicitly cast from `storage` arrays.

```
function function_3(uint _arrSize) public {
    uint[5] memory arrSizeFive;
    uint[] memory arrWithCustomSize = new uint[](_arrSize);
    uint[] memory localCopyOfArr = arr;
    // ...do something
}
```

The declaration pattern impacts gas cost, though keep in mind that the first two examples are empty, and would cost additional gas depending on how they are eventually filled.

```
function declareMemoryArrays() public view {
    uint[5] memory simpleArr; // this line costs 135 gas
    uint[] memory emptyArr = new uint[](5); // This line costs 194 gas
    uint[] memory arrCopy = arr; // This line costs 13166 gas
}
```

The lack of dynamic `memory` arrays can require some gymnastics if you need to create an array where the size is not initially known. Depending on the specific needs of the problem, valid solutions for filtering an array and returning a smaller array could include:

- Looping through a larger array twice, first to count the number, then to copy the appropriate elements
- Tracking the number of elements that meet condition X with a storage variable, then instantiating the array with `<type>[] memory filteredArray = new <type>[](numX);`
- Using multiple data structures to track references to different subsets

Calldata Arrays

Arrays in `calldata` are read only. Otherwise, they function the same as any other array.

Array **slices** are currently only implemented for `calldata` arrays.

Conclusion

In this lesson, you've learned the differences between the `memory`, `storage`, and `calldata` data locations. You've also learned how they apply to arrays, with each

having its own properties, restrictions, and costs.

Was this page helpful? Yes No Suggest edits Raise issue

Writing Arrays

Filtering Arrays