**Solidity Reference Types**

**Table of Contents**

* **Solidity Arrays**
* **Fixed Arrays**
* **Dynamic Arrays**
* **Bytes and Strings as Arrays**
* **Array Slicing**
* **Structs**
* **Mapping Types**

**Solidity Reference Types**

It is possible to modify the values of reference types by using a variety of different names. This is in contrast to value types, where each time a value type variable is used, you get an independent duplicate of that variable. This necessitates the fact that reference types must be handled with greater care than value types. At the moment, reference types include structs, arrays, and maps, among other things. If you use a reference type, you must always explicitly specify the data area in which the type is stored: memory (whose lifetime is limited to the duration of an external function call), storage (the location where the state variables are stored, whose lifetime is limited to the duration of a contract), or calldata (the location where the type is stored) (special data location that contains the function arguments).

The automated copy process will always be triggered by an assignment or type conversion that changes the data location, whereas assignments inside the same data location will only copy in some situations, depending on the storage type.

Data Locations:

Every reference type contains an additional descriptor, known as the "data location," that indicates where the data is kept. Memory, storage, and calldata are the three types of data storage available. Unlike memory, calldata is a static area that cannot be modified and is not persistent. It is used to store function arguments and acts in most ways like memory.

Storage vs Memory vs Calldata:

When designing smart contracts in Solidity, it is important to be aware of how your variables and data are handled by the Ethereum Virtual Machine (EVM). The decisions you make will have an impact on a variety of factors, including gas costs — whether to call your functions or deploy your contract — as well as the structure of your storage facility.

Given that every piece of block space in Ethereum is extremely valuable (hence the high price of Eth, which has since dropped), the efficiency of your code and the resulting contract, which will hopefully be invoked on a regular basis, is greatly influenced. The gas used by every function call adds up; every amount of savings adds up when summed across the potential lifetime of all function invocations in a given time period.

Storage:

Storage is the most straightforward concept to grasp because it is where all state variables are kept. Because the state of a contract can be changed (for example, within a function), storage variables must be mutable in order to be used. Their position, on the other hand, is permanent, and they are maintained on a blockchain system.

Whenever possible, several values of state variables will occupy the same storage slot in order to maximize storage efficiency. Aside from the exceptions such as dynamically sized arrays and structs, all other variables are compressed into blocks of 32 bytes.

If the total size of these variables is less than 32 bytes, they will be concatenated to occupy the same slot as the previous variable. If this is not the case, they will be moved to the next available storage slot. The data is stored consecutively (that is, one after the other), beginning with the 0 slot (and progressing to slots 1, 2, 3, and so on), in the sequence in which they were declared in the contract.

In all cases, dynamic arrays and structs take up a new storage slot, and any variables that come after them will be initialized to take up a new storage slot as well.

Because the size of both dynamic arrays and structs is unknown a priori (i.e. until they are assigned later in your contract), they cannot be stored with their data in between other state variables as they would otherwise be possible. Rather than taking up 32 bytes, they are assumed to take up 32 bytes, and the components contained within them are stored starting at a different storage slot that is computed using a Keccak-256 hash, as described above.

Constant state variables, on the other hand, are not saved into a storage slot. Instead, they are injected directly into the contract bytecode, and anytime those variables are read, the contract immediately switches them out for the constant value that was assigned to them in the beginning.

Memory

Variables specified within the scope of a function are given priority in terms of memory allocation. They only exist for the duration of the function call, and as a result, they are transient variables that cannot be retrieved outside of the function call (i.e. anywhere else in your contract besides that function). They are, however, modifiable inside the scope of that function.

There are four 32-byte slots for memory in Solidity, each with a specific bytes-per-byte range, which are as follows: 1) 64 bytes of scratch space for hashing methods; 2) 32 bytes of currently allocated memory size, which is the free memory pointer where Solidity always places new objects; and 3) a 32-byte zero slot — which is used as the initial value for dynamic memory arrays and should never be written to.

It is because of these design variations that there are instances where arrays and structs will use varying amounts of space depending on whether they are in storage or memory.

| uint8[4] arr; struct Str {  uint v1;  uint v2;  uint8 v3;  uint8 v4; } |
| --- |

In this example,Each of the array arr and the struct Str takes up 128 bytes of memory in each of the two circumstances (i.e. 4 items, 32 bytes each). However, as storage, arr merely takes up 32 bytes (1 slot), whereas Str takes up 96 bytes (3 slots, 32 bytes each).

Calldata:

Calldata is an immutable, temporary storage place where function arguments are kept, and it operates largely like memory in terms of performance and responsiveness.

It is suggested to make use of calldata wherever possible because it avoids the creation of unneeded copies and assures that the data is not updated. Return values from functions can additionally include arrays and structs with the calldata data location.

ABI specifies a format for this sort of data, which is considered to be in multiples of 32 bytes and in accordance with the ABI definition (which differs from internal function calls). The arguments for constructors are a little different, in that they are appended directly to the end of the contract's code instead of being prefixed with the contract's name (also in ABI encoding).

Comparisons

Once a reference type variable (array or struct) is defined, the data location for that variable must also be specified unless the variable is of type state, in which case it is automatically interpreted as storage. Since Solidity v0.6.9, memory and calldata are permitted in any functions, regardless of whether or not they are publicly visible (ie external, public, etc).

Similarly to objects or arrays in Javascript, assignments will either result in copies of the data being generated or simply references to the same piece of data being created:

Assignments between storage and memory (or from calldata) always result in the creation of a new copy.

Assignments from one memory to another only result in the creation of references. As a result, altering one memory variable has an effect on all other memory variables that refer to the same data as the changed memory variable.

Assignments from storage to a local storage variable also only assign a reference to the storage variable in question.

Everything else that is assigned to storage is always copied.

It is advised to use calldata rather than memory when passing array parameters into functions because this results in significant gas savings. For example, using calldata in a summing function that loops over an input array can save about 1829 gas (or 3.5 percent) on average.

Comparisons:

Once a reference type variable (array or struct) is defined, the data location for that variable must also be specified unless the variable is of type state, in which case it is automatically interpreted as storage. Since Solidity v0.6.9, memory and calldata are permitted in any functions, regardless of whether or not they are publicly visible (ie external, public, etc).

Similarly to objects or arrays in Javascript, assignments will either result in copies of the data being generated or simply references to the same piece of data being created:

* Assignments between storage and memory (or from calldata) always result in the creation of a new copy.
* Assignments from one memory to another only result in the creation of references. As a result, altering one memory variable has an effect on all other memory variables that refer to the same data as the changed memory variable.
* Assignments from storage to a local storage variable also only assign a reference to the storage variable in question.
* Everything else that is assigned to storage is always copied.

It is advised to use calldata rather than memory when passing array parameters into functions because this results in significant gas savings. For example, using calldata in a summing function that loops over an input array can save about 1829 gas (or 3.5 percent) on average.

| // Gas used: 50992 function func1 (uint[] memory nums) external {  for (uint i = 0; i < nums.length; ++i) {  ...  } } // Gas used: 49163 function func2 (uint[] calldata nums) external {  for (uint i = 0; i < nums.length; ++i) {  ...  } } |
| --- |

**Solidity Arrays**

A data structure that stores a fixed-size sequential collection of objects of the same type is known as an array data structure. The term "array" refers to a collection of data that is stored in a single location, however it is often more practical to conceive of an array as a collection of variables with the same type.

Instead of declaring individual variables such as number0, number1,..., and number99, you declare a single array variable such as numbers and use the values numbers[0], numbers[1], and..., numbers[99] to represent individual variables, as seen in the following example. The index of a specific element in an array is used to access that element.

In Solidity, an array can have a fixed size at compile time or can have a dynamic size. There are many different sorts of elements that can be used in a storage array. In the case of a memory array, the element type cannot be mapping, and if the element type is to be used as a function argument, the element type must be of the ABI class.

All arrays are made up of memory addresses that are contiguous. The first element is represented by the lowest address, while the last element is represented by the highest address.

Declaring Arrays:

If the programmer wants to declare an array of fixed size in Solidity, he or she must specify the type of elements and the number of elements required by the array in the following ways:

| type arrayName [ arraySize ]; |
| --- |

This is called a single-dimension array. Array type can be any acceptable Solidity data type, and arraySize must be an integer constant more than zero. For example, the following statement can be used to declare a 10-element array of type uint named balance:

| uint balance[10]; |
| --- |

If the programmer wants to declare an array of dynamic size in Solidity, he or she must first specify the type of the components as follows:

| type[] arrayName; |
| --- |

Initializing Arrays:

The following statement can be used to initialize Solidity array elements in a single statement:

| uint balance[3] = [1, 2, 3]; |
| --- |

The number of values included within the braces [] cannot be greater than the number of items contained within the square brackets [] that we specify for the array. The following is an example of how to assign a single element of an array to the variable

If the size of the array is not specified, an array that is only large enough to hold the initialization is generated. As a result, if you write, you are correct.

| uint balance[] = [1, 2, 3]; |
| --- |

## Creating dynamic memory arrays:

Dynamic memory arrays are created using new keyword.

| uint size = 3; uint balance[] = new uint[](size); |
| --- |

## 

## Accessing Array Elements:

An element is accessible by indexing the array name into the element's position. This is accomplished by putting the index of the element between square brackets after the name of the array after the name of the array. As an illustration,

| uint salary = balance[2]; |
| --- |

Using the preceding line, the third member of the array will be taken and assigned to the salary variable. The following is an example that will make use of all three of the previously described principles, namely declaration, assignment, and accessing arrays.

Array Members:

length − length returns the size of the array. Length can be used to change the size of the dynamic array by setting it.

push() − push allows you to append an element to a dynamic storage array at the end. It returns the new length of the array.

push(x):Dynamic storage arrays and bytes (not string) have a member function called push(x) that you can use to append a given element at the end of the array. The function returns nothing.

pop():Dynamic storage arrays and bytes (not string) have a member function called pop() that you can use to remove an element from the end of the array. This also implicitly calls delete on the removed element.

Important:

Increasing the length of a storage array by calling push() has constant gas costs because storage is zero-initialized, while decreasing the length by calling pop() has a cost that depends on the “size” of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling delete on them.

The following code explains various array properties and methods:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.10;  contract Array {  // Several ways to initialize an array  uint[] public arr;  uint[] public arr2 = [1, 2, 3];  // Fixed sized array, all elements initialize to 0  uint[10] public myFixedSizeArr;   function get(uint i) public returns (uint) {  return arr[i];  }   // Solidity can return the entire array.  // But this function should be avoided for  // arrays that can grow indefinitely in length.  function getArr() public returns (uint[] memory) {  return arr;  }   function push(uint i) public {  // Append to array  // This will increase the array length by 1.  arr.push(i);  }   function pop() public {  // Remove last element from array  // This will decrease the array length by 1  arr.pop();  }   function getLength() public returns (uint) {  return arr.length;  }   function remove(uint index) public {  // Delete does not change the array length.  // It resets the value at index to it's default value,  // in this case 0  delete arr[index];  }   function examples() external {  // create array in memory, only fixed size can be created  uint[] memory a = new uint[](5);  } } |
| --- |

Bytes and string as Arrays:

Arrays of the types bytes and string are special types of arrays. In many ways, the bytes type is identical to bytes1[], except it is compressed into a smaller amount of calldata and memory. Strings are equal to bytes, except they do not enable access to the length or index of the string.

Despite the fact that Solidity does not include any string manipulation functions, there are third-party string libraries available. Another useful function is keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2), which compares two strings based on their keccak256-hash. You can also concatenate two strings by using string.concat (s1, s2).

Use bytes rather than bytes1[] since it is less expensive. Using bytes1[] in memory adds 31 padding bytes between the elements, thus you should use bytes instead. It should be noted that the padding is absent in storage as a result of the tight packing (see bytes and string). When dealing with arbitrary-length raw byte data, bytes should be used and string should be used when dealing with arbitrary-length string (UTF-8) data. In cases when you are able to restrict the length to a specific number of bytes, always use one of the value types bytes1 to bytes32 because they are far more cost-effective.

Array Slicing:

When seeing an array slice, you are looking at a continuous section of the array. They are denoted by the notation x[start:end], where start and end are expressions that result in the type uint256 (or implicitly convertible to it). The initial element of the slice is represented by x[start], while the last piece is represented by x[end - 1.]

An exception is thrown if start is greater than end, or if end is greater than the length of the array, as the case may be.

Neither the start nor the end are required; start defaults to 0 and end defaults to a value equal to the length of the array.

Array slices do not have any members in their arrays. In addition to supporting index access, they are implicitly convertible to arrays of their underlying type. The index access in the underlying array is not absolute, but rather relative to the beginning of the slice.

Array slices do not have a type name, which means that they cannot be used as a type in a variable; instead, they can only be found in intermediate expressions.

| bytes exampleBytes = '0xabcd' exampleBytes[2:5]; # 'abc'  exampleBytes[:5]; # '0xabc'  exampleBytes[2:]; # 'abcd'  exampleBytes[:]; # '0xabcd' |
| --- |

**Structs**

Record representation is accomplished through the usage of structs in Solidity. Consider the following scenario: you want to keep track of your books in a library. You may want to keep track of the following characteristics about each book:

* Title
* Author
* Subject
* Book ID

## Defining a Struct:

The struct keyword must be used in order to define a Struct. The term struct is used to define a new data type that has more than one member. The following is the format of the struct statement:

| struct struct\_name {   type1 type\_name\_1;  type2 type\_name\_2;  type3 type\_name\_3; } |
| --- |

Example:

| struct Book {   string title;  string author;  uint book\_id; } |
| --- |

## 

## 

## Accessing a Struct and its variable:

To access any member of a structure, we use the member access (.) operator, which lets us get to any part (.). It looks like a period between the structure variable name and the structure member that we want to get to. You would use the struct to set up variables of the structure type with the help of this tool. Structures can be used in programs like this one to make things easier for people to understand.

The Following code will help you understand how structs work in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.10;  contract Todos {  struct Todo {  string text;  bool completed;  }   // An array of 'Todo' structs  Todo[] public todos;   function create(string memory \_text) public {  // 3 ways to initialize a struct  // - calling it like a function  todos.push(Todo(\_text, false));   // key value mapping  todos.push(Todo({text: \_text, completed: false}));   // initialize an empty struct and then update it  Todo memory todo;  todo.text = \_text;  // todo.completed initialized to false   todos.push(todo);  }   // Solidity automatically created a getter for 'todos' so  // you don't actually need this function.  function get(uint \_index) public returns (string memory text, bool completed) {  Todo storage todo = todos[\_index];  return (todo.text, todo.completed);  }   // update text  function update(uint \_index, string memory \_text) public {  Todo storage todo = todos[\_index];  todo.text = \_text;  }   // update completed  function toggleCompleted(uint \_index) public {  Todo storage todo = todos[\_index];  todo.completed = !todo.completed;  } } |
| --- |

**Mapping**

Mapping is like a hash table or a dictionary in any other language when you use it in Solidity . A key can be one of any of the built-in types, but reference types aren't allowed. The value can be any type, too. Mappings are mostly used to link an Ethereum address to a value type.

Syntax:

| mapping(key => value) <access specifier> <name>; |
| --- |

### Creating a Mapping:

Mapping is defined as any other variable type, which accepts a key type and a value type.

Mapping Example 1

| // Solidity program to  // demonstrate mapping pragma solidity ^0.8.0;    // Defining contract  contract mapping\_example {    //Defining structure  struct student   {  // Declaring different   // structure elements  string name;  string subject;  uint8 marks;  }  // Creating a mapping  mapping (  address => student) result;  address[] public student\_result;  } |
| --- |

Mapping Example 2: Mapping and nested mapping

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.10;  contract Mapping {  // Mapping from address to uint  mapping(address => uint) public myMap;   function get(address \_addr) public returns (uint) {  // Mapping always returns a value.  // If the value was never set, it will return the default value.  return myMap[\_addr];  }   function set(address \_addr, uint \_i) public {  // Update the value at this address  myMap[\_addr] = \_i;  }   function remove(address \_addr) public {  // Reset the value to the default value.  delete myMap[\_addr];  } }  contract NestedMapping {  // Nested mapping (mapping from address to another mapping)  mapping(address => mapping(uint => bool)) public nested;   function get(address \_addr1, uint \_i) public returns (bool) {  // You can get values from a nested mapping  // even when it is not initialized  return nested[\_addr1][\_i];  }   function set(  address \_addr1,  uint \_i,  bool \_boo  ) public {  nested[\_addr1][\_i] = \_boo;  }   function remove(address \_addr1, uint \_i) public {  delete nested[\_addr1][\_i];  } } |
| --- |