Array

```
contract Array {
    uint8[3] public fixedU8Array;
    uint8[] public dynamicU8Array;

    uint256[3] public fixedU256Array;
    uint256[] public dynamicU256Array;
}

/*
*   Array Variable    |   Slot(s) Occupied    |   Values Stored At Slot(s)
[sload(slot)]
*   __________________|_______________________|
_____________________________________________________
*   fixedU8Array      |   0 - 2               |   0 <= x <= 2
*   dynamicU8Array    |   3                   |   keccak256(3) + (0 <= x <
dynamicU8Array.length)
*   fixedU256Array    |   4 - 6               |   4 <= x <= 6
*   dynamicU256Array  |   7                   |   keccak256(7) + (0 <= x <
dynamicU256Array.length)
*   __________________|_______________________|
_____________________________________________________
*/
```

Array Storage

Arrays in Yul take up slots depending on their array type. Array types include:

- Fixed Array, i.e. `fixedU256Array`
- Dynamic Array, i.e. `dynamicU256Array`

Because the length of a fixed arrays are known, they take up slots as much as their length. i.e. `fixedU256Array` is of length `3`, therefore, it will take up 3 slots starting from its current slot. However, slot taking of a fixed array is also controlled by its data type, uint256 will take up an entire slot. So there will be as many slots occupied as 3 `uint256` numbers in the `fixedU256Array`.

In the case of `fixedU8Array`, which has a known length of 3, because it's of type `uint8`, it will be packed in that slot where it should start from.

Dynamic arrays are different, because, their lengths are not known, so there is no set amount of slots allocated for it, therefore, at their slots, which we will call `slot`, the length of the array is stored. Then the array elements can be found starting at slot `keccak256(slot)` and spanning up till `keccak256(slot)` + ((the length of the array) - 1).

If dynamicU256Array should have 5 elements, at its slot in storage, which we will assume as `c`, the value `5` will be stored. Then to get these 5 elements, they will be found at:

Index 0 => `keccak256(c) + 0`
Index 1 => `keccak256(c) + 1`
Index 2 => `keccak256(c) + 2`
Index 3 => `keccak256(c) + 3`
Index 4 => `keccak256(c) + 4`

All 5 elements. Adding one element to the array will increment the value at slot `c` to `6` and also add an:

Index 5 => `keccak256(c) + 5`

Which is the `6th` element.

Reading From An Array

While reading the elements of the `fixedU256Array` is simple, because, type `uint256` takes up 32 bytes, the same cannot be said for the `fixexU8Array` as they're packed and care would be taken to avoid messing up the value stored at that slot. We can always call an `sload(slot)` to get the value of any index stored at any slot.

However, for `fixedU8Array` on initialization:

```
contract Array {
    uint8[3] public fixedU8Array;

    constructor() {
        // Initialize array variables.
        fixedU8Array = [1, 2, 3];
    }
}
```

Once the slot where the values of the `fixedU8Array` has been located, which we shall address as `loc`, we can call a simple `sload(loc)` to get the value stored at that slot, which will be returned as:

```
0x0000000000000000000000000000000000000000000000000000000000030201
```

It will be packed, as stated earlier. We can then use a masking to return any index in the array based of the `uint8` type, `1 byte` by calculating its offset precisely.

More Info

Click to read more.

Bitwise

What Are Bitwise Operations?

Understanding Bitwise Operators

Bitwise Operators:

Name	Symbol
AND	&
LEFT SHIFT	<<
NOT	~
OR	|
RIGHT SHIFT	>>
XOR	^

More Info

Understanding Bitwise Operators Bitwise Operators Wiki

Call

Calldata encoding: https://rb.gy/vmzhck.

The callContract function in the CallerContract contract calls the setNumber function in the CalledContract contract via a call opcode. This opcode takes in 7 arguments: `gas`
`address`
`value`
`dataOffset`
`dataSize`
`returnDataOffset`
`returnDataSize`

gas

Amount of gas to send in `call`, usually set to `gas()` or any specified number, `n`.

address

Address to make `call` to. If the address does not have a function that matches the identifier, the `fallback` function is called. If it doesn't have one, it returns `false`.

`NOTE`: `Call`s made to invalid addresses return true, `call` returns false on two occassions:

1. The address has no `fallback` function.
2. The function called reverts.

value

Amount in ETH to be sent.

dataOffset and dataSize

The `dataOffset` and `dataSize` are determined by the size of the encoded calldata sent to the `adderss`.

First, we store the function selector, `3fb5c1cb`, which we already have as a literal in location `0x00`, setting location `0x00` to `0x1f` to:

```
0x0000000000000000000000000000000000000000000000000000000003fb5c1cb
```

NOTE: If we store the selector in a variable:

```
bytes4 selector = 0x3fb5c1cb;
```

and then save it to location 0x00, we will have:

```
0x3fb5c1cb0000000000000000000000000000000000000000000000000000000000
```

`PS:` FIXED AND DYNAMIC BYTE ARRAYS ARE LEFT ALIGNED, EVERYTHING ELSE IS RIGHT ALIGNED.

We can then encode the number we want to pass, takes as `4`, in the next location `0x20 - 0x3f`, leaving us with:

```
0x0000000000000000000000000000000000000000000000000000000000000004
```

In location `0x20 - 0x3f`. We can then calculate the offset and size of bytes we want to pass through in our `call` opcode by observing the location of `3f` in location `0x00 - 0x1f`, getting the size in bytes from that point to `0x1f`, and then adding it to `32` bytes, which is the size of the data stored in location `0x20 - 0x3f`.

The location of `3f` is `0x1c`, our `dataOffset`.

This will leave us with `0x1c` to `0x1f`, which is 4 bytes, and `0x20 - 0x3f` which is 32 bytes. 32 bytes + 4 bytes = `36 bytes`, our data size.

```
let success := call(gas(), _called, 0, 0x1c, 0x24, 0, 0)
// 0x1c == Offset.
// 0x24 == 36 bytes.
```

`returnDataOffset` and `returnDataSize`

Specifies the offset and size of the data returned from the call to copy to memory. The returnDataSize is obtained via the `returndatasize()` opcode in Yul. The offset is any offset specified by the developer to match his desired returned value. If there is no returned data or the returned data is not needed, `0x00` or `0` can be passed for both cases.

```
let success := call(gas(), _called, 0, 0x1c, 0x24, 0, 0)
```

Calldata Encoding

`NOTE: m` => memory location, `c` => calldata location (which is still memory, but dedicated at the moment).

This file is used to elaborate how the arguments encoded in the src/Calldata.sol file have been encoded for all functions defined in the CallerContract contract.

Function 1:

`callAdd`

Assume num = 7

This calls the `add(uint256)` function with selector `0x1003e2d2`. The calldata arguments for the `add` function can fit into the `scratch space` i.e. first 64 bytes of memory, `0x00` to `0x5f`. So we can comfortably fill it between `0x00` and `0x5f`.

Encoding

```
c : ---------- | m : 0x00 - 0x1f | 4 bytes, starting from 1f
=> 0x0000000000000000000000000000000000000000000000000000000001003e2d2
=> Function Selector (This part of the memory is not included as a starting
point in the calldata, but a place to store the selector of the function to
call).

c : 0x00 - 0x1f | m : 0x20 - 0x3f | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000007
=> Hex encoding of the number we want to pass as argument to the function
selector.

Total: 36 bytes | 36 | 0x24
```

Function 2:

`callMultiply`

Assume num1 = 6, num2 = 7

This calls the `multiply(uint8,uint8)` function with selector `0x6a7a8e0b`. The data arguments we need will not be contained in the `scratch space` as they're `the selector`, `num1` and `num2`, all 32 bytes. We will need to encode them in the free memory, which starts from `0x80`. Also, despite us having `uint8` types which would mean `0xff`,

the `ABI` demands we pass them as 32 bytes as well.

Encoding

```
c : ---------- | m : 0x80 - 0x9f | 4 bytes, starting from 9f
=> 0x000000000000000000000000000000000000000000000000000000006a7a8e0b

c : 0x00 - 0x1f | m : 0xa0 - 0xbf | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000006
=> Hex encoding of the first argument to the function selector.

c : 0x20 - 0x3f | m : 0xc0 - 0xdf | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000007
=> Hex encoding of the second argument to the function selector.

Total: 68 bytes | 68 | 0x44
```

Function 3

`callArraySum`

Assume num1 = 1, num2 = 2, num3 = 3, num4 = 4

This calls the `arraySum(uint256[])` function with selector `0x7c2b11cd` with 4 array values.

Encoding

```
c : -----------     | m : 0x80 - 0x9f       | 4 bytes, starting from 9f
=> 0x0000000000000000000000000000000000000000000000000000000007c2b11cd

c : 0x00 - 0x1f     | m : 0xa0 - 0xbf       | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000020
=> Array pointer (points to 0x20 in calldata, the length of the array is
stored there).

c : 0x20 - 0x3f     | m : 0xc0 - 0xdf       | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000004
=> Length of array.

c : 0x40 - 0x5f     | m : 0xe0 - 0xff       | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000001
=> First number, 1.

c : 0x60 - 0x7f     | m : 0x0100 - 0x011f   | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000002
=> Second number, 2.

c : 0x80 - 0x9f     | m : 0x0120 - 0x013f   | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000003
=> Third number, 3.

c : 0x0xa0 - 0xbf   | m : 0x0140 - 0x015f   | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000004
=> Fourth number, 1.

Total: 196 bytes | 196 | 0xc4
```

Function 4

`callSetString`

Assume str = "Hello"

This calls the `setString(string)` function with selector `0x7fcaf666` with 4 array values. However, this might get tricky. Since the memory can only contain one 32 bytes value at a time, we might make some impositions that will restrict the set string to 32 characters. However, since we want to set a storage string in the `CalledContract`, the max a storage string can take is 31 bytes, so we will restrict our function to only take in 31 bytes. These first two lines will handle that for us. Having strings with len > 31

```
function callSetString(string calldata str) public {
    uint8 len = uint8(bytes(str).length);
    if (len > 31) revert();
    _;
}
```

`NOTE` : Whenever a memory reference is passed inside a function, it advances the free memory pointer by 64 bytes. Memory references includes: `abi.encode()`, `abi.encodePacked()`, `bytes memory`, `string memory`, `bytes constant` (Storage variable, this will advance the pointer by 128 bytes because the value will be substituted and read again).

It is advised to `ALWAYS` use `mload(0x40)` to write to the next slot then update to avoid accidental overwrites.

Encoding

```
function callSetString(string calldata str) public {
    _;
    bytes memory strCopy = bytes(str);
    _;
}
```

This moves the memory by 64 bytes, which we will store at a random location we know won't be overwritten until the function call is over `0x0200`.

```
c : ----------     | m : 0x80 - 0x9f       | 4 bytes, starting from 9f
=> 0x0000000000000000000000000000000000000000000000000000007fcaf666

c : 0x00 - 0x1f     | m : 0xa0 - 0xbf       | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000020
=> String pointer (points to 0x20 in calldata, the length of the string is
stored there).

c : 0x20 - 0x3f     | m : 0xc0 - 0xdf       | 32 bytes
=> 0x0000000000000000000000000000000000000000000000000000000000000005
=> Length of string "Hello".

c : 0x40 - 0x5f     | m : 0xe0 - 0xff       | 32 bytes
=> 0x48656c6c6f000000000000000000000000000000000000000000000000000000
=> String Hello in bytes.

Total: 100 bytes | 100 | 0x64
```

`PS:` FIXED AND DYNAMIC BYTE ARRAYS ARE LEFT ALIGNED, EVERYTHING ELSE IS RIGHT ALIGNED.

Function 5

`callStructCall`

Assume num1 = 6, num2 = 7

The encoding is similar to the `callMultiply`.

Conditionals

There are no parentheses in the if statements, and there are no elses, rather, use switch.

```
0 => False
1 => True
```

If

```
if lt(x, 10) { res := 0 }
if gt(x, 10) { res := 1 }
```

Switch

```
switch lt(x, 79)
case 1 { isTrue := 0x01 }
default { isTrue := 0x00 }
```

Counter

Study: https://rb.gy/3wj60

Yul does not have any checks for over and underflow, therefore, whenever the `inc()` is called, the function asserts that the current number is not equal to the max of uint256,

```
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

And then, increments it by one.

Also, calling the `dec()` will assert that the current value is not equal to `0`, and then, decrements it by one.

Enums

`enum` s are treated as `uint8` , hence, packed.

```
myEnum myenum = myEnum.STOPPED;
```

Errors

Errors are encoded the same way a calldata is encoded. First their 4 byte selector, then their arguments. And are read starting from the start of the selector with a size as large at their selector + encoded arguments.

```
if lt(errorNumber, mainNumber) {
    mstore(0x00, errorSelector) // 4 bytes.
    mstore(0x04, storedNumberForError) // 32 bytes.
    revert(0x00, 0x24) // Reads 36 bytes.
}
```

Errors that do not want to return data simply revert with `(0x00, 0x00)`.

```
if lt(errorNumber, mainNumber) {
    revert(0x00, 0x00)
}
```

EtherWallet

On Deployment

`constructor` is marked `payable`, and will accept payments >= 0 ETH to deploy contract. In this case, it retrieves the address of `msg.sender` from the `caller()` opcode and saves it in slot `0` (the `owner` variable). `receive()` allows ETH sent from external sources.

getBalance()

To get the balance of the contract, there are two ways to go about that:

1. `balance(address())` `address()` returns the address of the contract in execution, `address(this)`. `balance(address _address)` returns the balance of ETH at `_address`.
2. `selfbalance()` This is an easier way to get the balance of the executing contract.

withdraw()

It evaluates that `caller()` is the owner of the contract by comparing the value of `caller()` with the value stored at slot `0`. If they match, the `_amount` passed is withdrawn via `call`. If they don't, it stores the `UnauthorizedSelector` at location `0x00`, and reverts, returning the first 4 bytes of the error selector.

Events

Events are emitted through the `log()` opcode which can allow up to `4` indexed events (for `anonymous events`) and `3` indexed events (for `non-anonymous events`). They range from `log0` to `log4` with each having a default `offset` and `size` as their first two parameters. This `offset` and `size` is the start and size of the `ABI encoded data` of the event's `unindexed` arguments. If there are no `unindexed` values, then `0x00` is stored for both `offset` and `size`.

Refer to https://rb.gy/bf8b1.

If the event is a `non-anonymous event`, the first topic, `topic[0]` is the `event signature`, then the subsequent topics, `topic[1] - topic[3]` are the indexed values of that event.

If the event is an `anonymous event`, there is no `event signature` stored, just the indexed event values.

Fallback

This is a function called when the `function selector` sent as a transaction to the contract doesn't match any function in that contract.

It is specified with a:

```
fallback() external {}
```

OR

```
fallback() external payable {}
```

In our Fallback contract, it sends `1 Wei` to any caller.

ForLoop

In Yul, there is only a `for` loop, and no `while` loop, however, we can rewrite a `for` loop to match a `while` loop.

```
for { let i := 0 } lt(i, 10) { i := add(i, 1) }
```

Is the Yul way of writing:

```
for (i = 0; i < 10; i++)
```

While Loop Imitation

Source: http://rb.gy/4narf.

```
assembly {
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {    // while(i < 256), 100 (hex) = 256
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Functions

Yul can have functions in them, and these functions can return values as well.

A Simple Sum Function Without Return Value

This function will take two numbers, add them, and then store the sum in location `0x00`.

```
assembly {
    function sum(num1, num2) {
        mstore(0x00, add(num1, num2))
    }
    sum(a, b)
}
```

A Simple Sum Function With Return Value

This function will take two numbers, add them, return the value as `total` and then store it in location `0x00`.

```
assembly {
    function sum(num1, num2) -> total {
        total := add(num1, num2)
    }
    mstore(0x00, sum(a, b))
}
```

PS : **BEWARE OF OVER AND UNDERFLOWS!!!**

Hash

Study http://rb.gy/8bqae.

```
49206f64206e746f20657078746563206f736d6f6e6565206f6877206574646e6f732065646e7
26e756173746420494142063696e676e64656f206f74207467652073687469206974637 06f2e
```

Hello World!

Source: http://rb.gy/j97db.

A simple Yul "Hello World!" code.

Returned string is `ABI Encoded`.

IsContract

This checks if an `address` is a contract or an `EOA` by looking at the size of the code. `EOA`s do not have code, `contracts` have code. The `extcodesize()` opcode returns the size of this code which is `0` for `EOA`s and `> 0` for `contracts`.

Mapping

Read about mapping storage here: http://rb.gy/87a8q.

SafeOperations

A version of Unchecked.sol with overflow checks.

SendEther

Sends ether to any address (including `address(0)`) using `call`.

SignatureVerification

Resource: http://rb.gy/akusx.

SimpleStorage

State variables are stored in locations called `slot`s. This code writes to a state variable slot using `sstore(slot, value)`, and reads from that particular slot using `sload(slot)`.

Structs

I honestly do not know how to explain what I have written for you to understand 😭. I hope you do.

Types

Showcasing the basic types `uint`, `string`, `address`, using Yul.

Unchecked

Yul `unchecked` default math operations.

YulERC20

A simple implementation of OpenZeppelin's `ERC20` contract, but with all functionalities written in Yul.