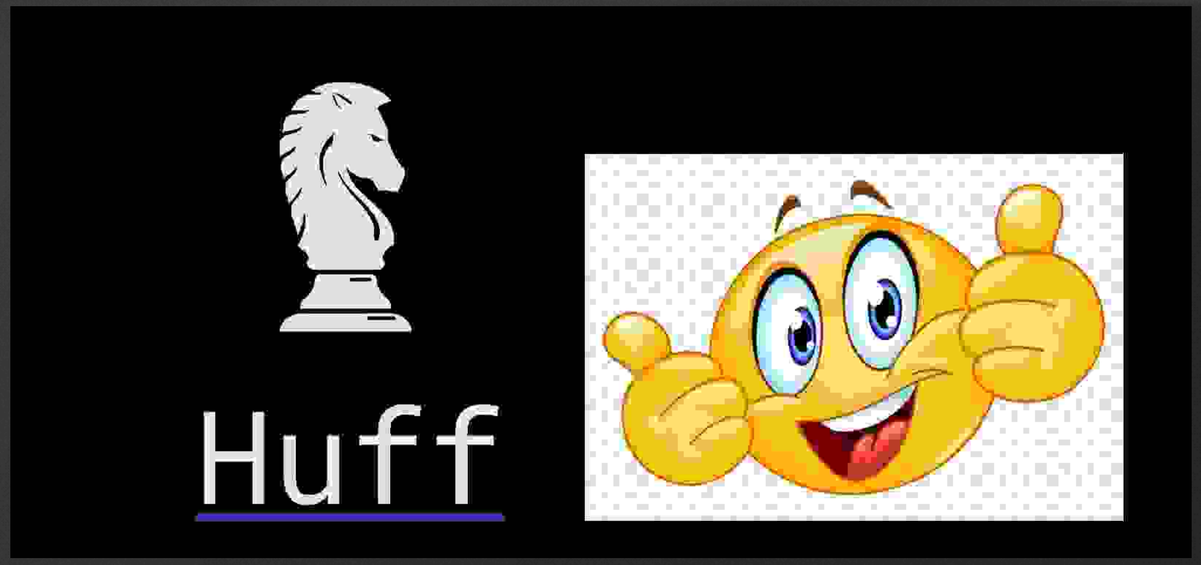
# What the huff

## Adding 2 and 2 together

makemake

# What is huff

- Huff is a low-level programming language for creating highly optimized smart contracts on the EVM.
- Huff does not hide the inner workings of the EVM and instead exposes its programming stack to the developer for manual manipulation.



# 1) What

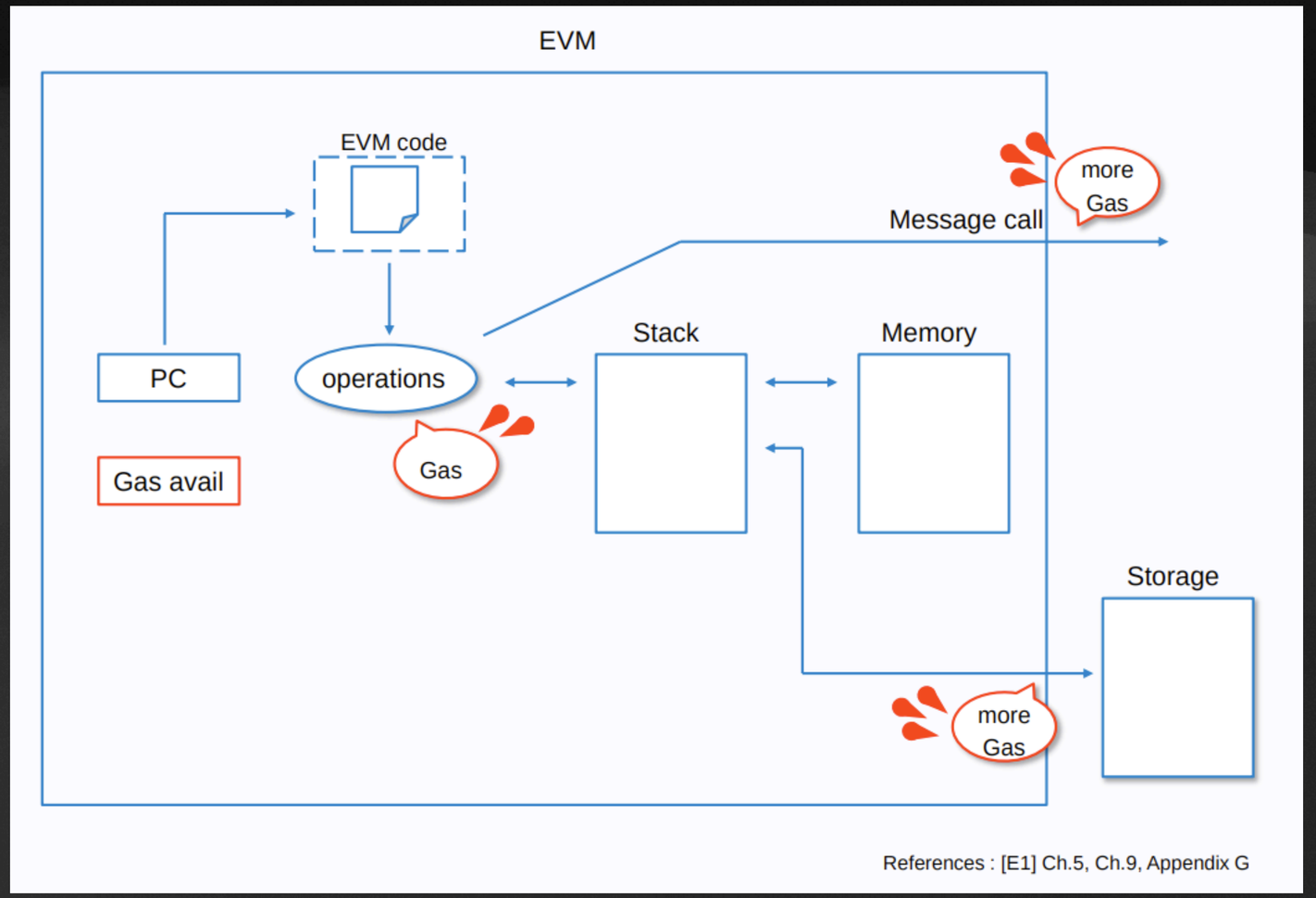
- Huff code looks like this.
- If it looks like evm bytecode, that’s because it is! Huff is essentially what we call an assembler.
- What is an assembler?

```
#include "huffmate/utils/Errors.huff"

#define macro ADD_TWO() = takes (2) returns (1) {
    // Input Stack:  [a, b]
    add           // [a + b]
    // Return Stack: [a + b]
}

#[calldata("0x0000000000000000000000000000000000000000000000000000000000000001"), value
#define test MY_TEST() = {
    0x00 calldataload   // [0x01]
    callvalue           // [0x01, 0x01]
    eq ASSERT()
}
```

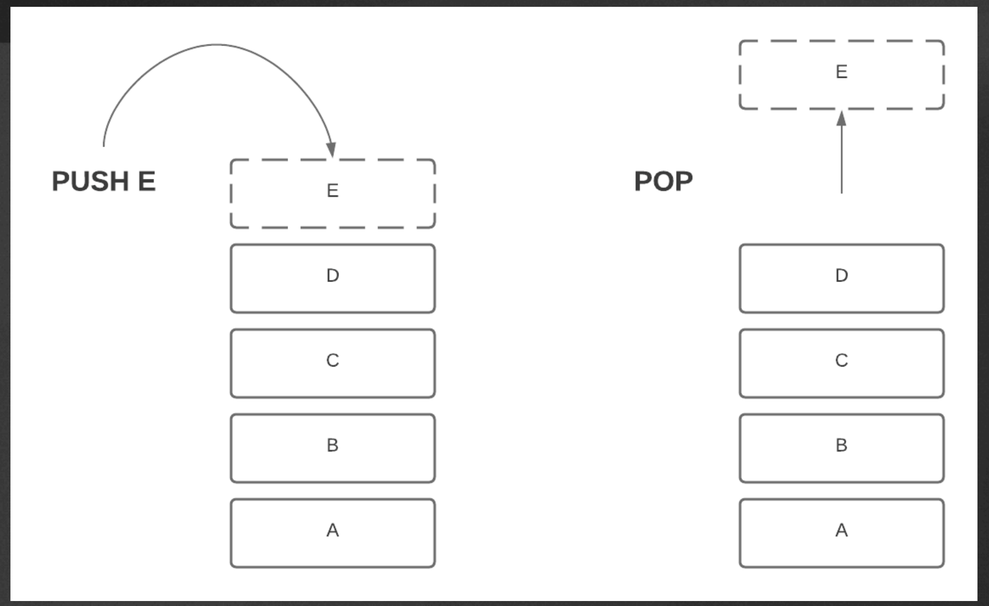
# What is the evm

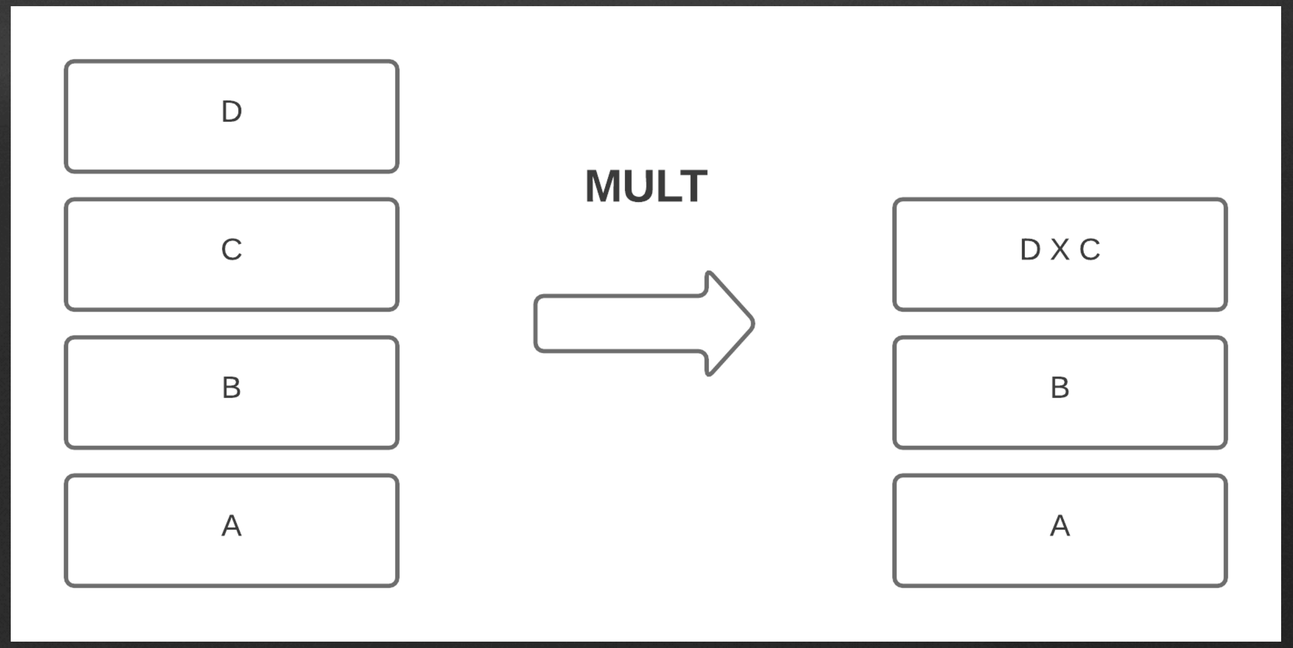
# Understanding the EVM

- The Ethereum Virtual Machine (EVM) is the computation engine behind Ethereum, similar to virtual machines in Microsoft's .NET Framework or interpreters for bytecode-compiled languages like Java.
- The EVM manages the deployment and execution of smart contracts, acting as a global decentralized computer with each contract having its own permanent data store.



- The EVM operates as a stack machine with a depth of 1024 items, each being a 256-bit word (32 bytes) compatible with 256-bit encryption. It uses stack operations like PUSH, POP, and applies instructions such as ADD and MUL to the top values.

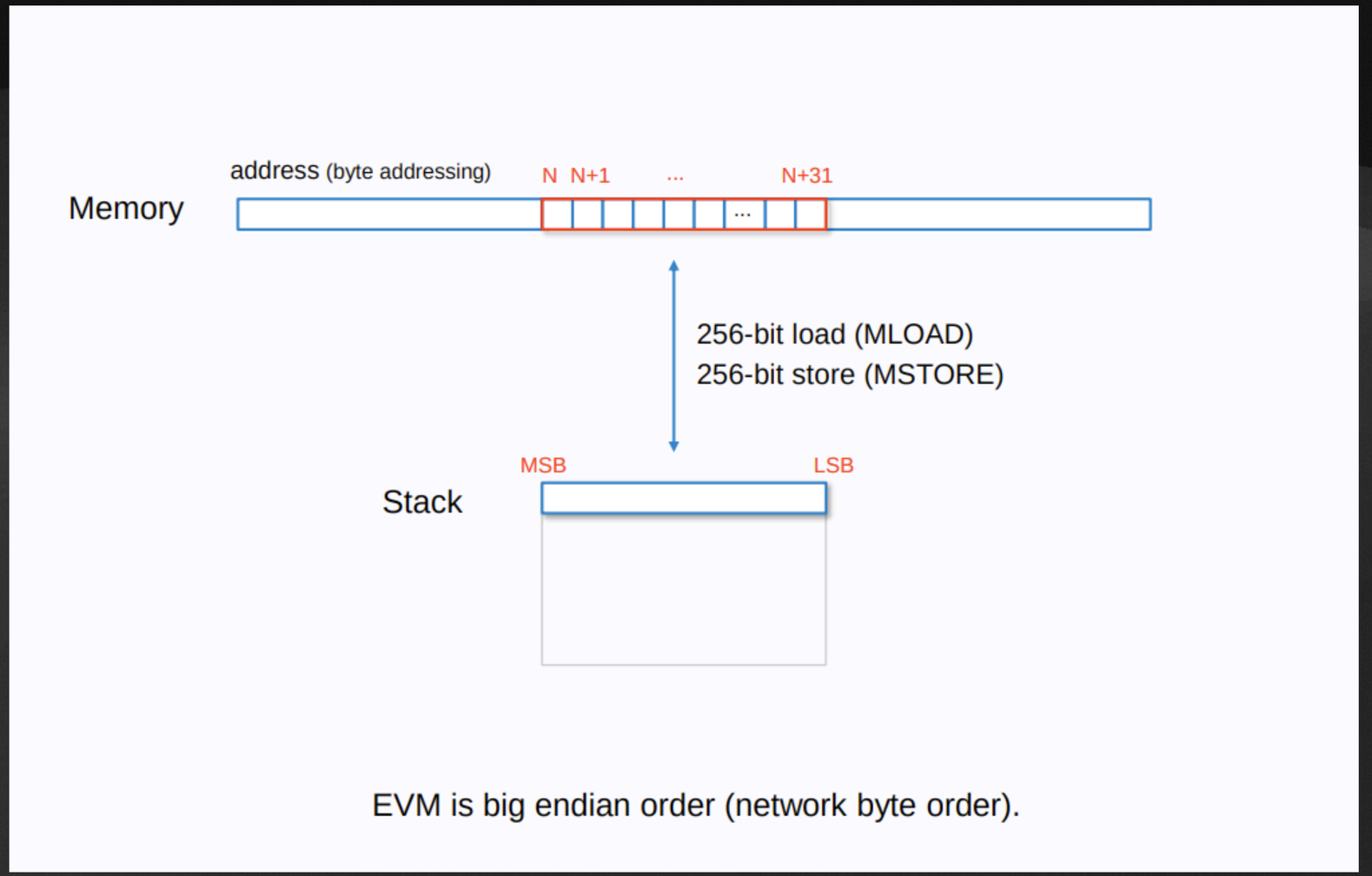



EVERYBODY GANGSTA
UNTIL THE CUTE PET
HUFF
imgflip.com

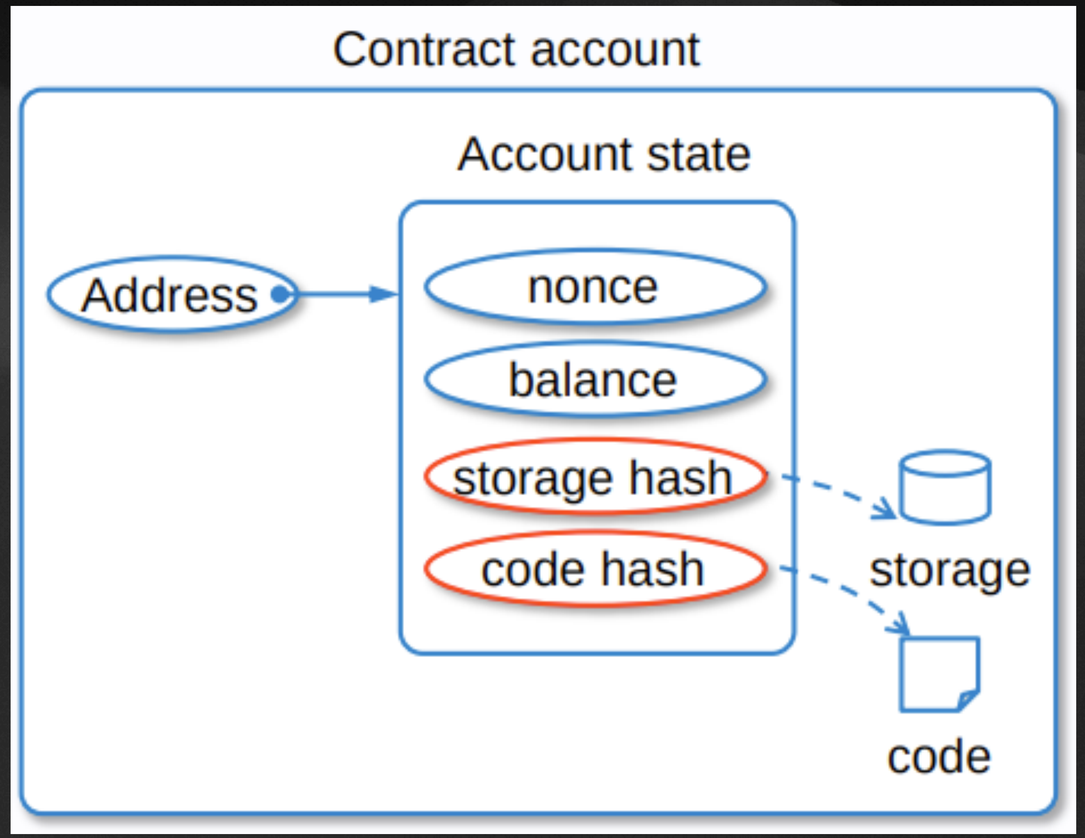
# Memory grindset

- Memory in the EVM is an expandable, byte-addressed, 1D array that starts empty and costs gas to read, write, and expand. Calldata, included in the transaction's payload, is similar but cannot be expanded or overwritten. Both read and write operations access or modify the first 256 bits (32 bytes) after a given pointer. Memory and calldata are volatile, being forgotten after the transaction finishes.



# Storage

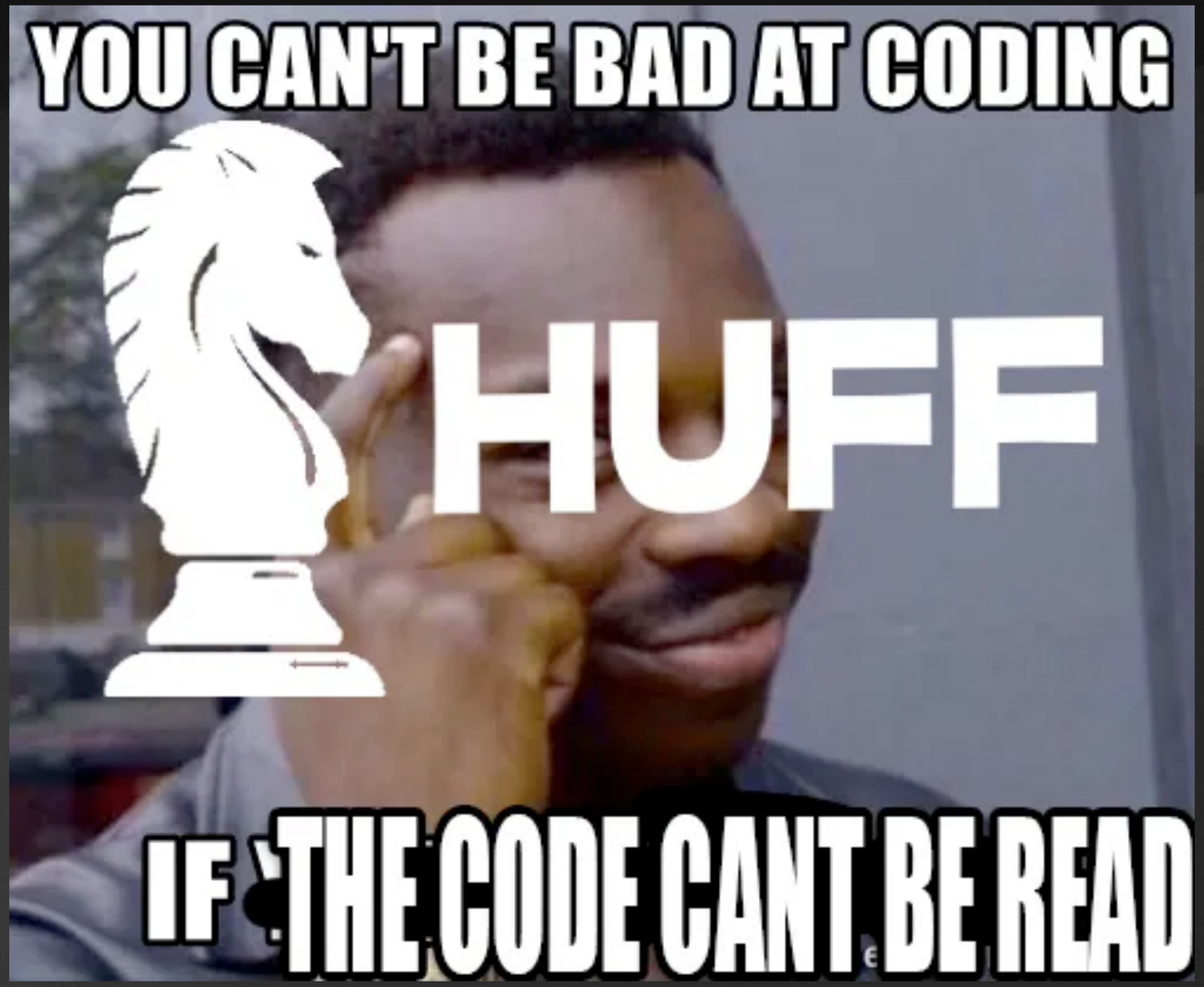
- Contract accounts on Ethereum store data persistently in a key-value store.
- Contract storage is more expensive to read and write than memory because all Ethereum nodes must update the contract's storage trie after a transaction.
- Storage can be viewed as a 256-bit to 256-bit map with 2^256 slots, unlike memory which is a large 1-dimensional array.



# Storage example

```
PUSH20 0xdEaDbEeFdEaDbEeFdEaDbEeFdEaDbEeFdEaDbEeF // [dead_addr]
PUSH1 0x00                                        // [0x00, dead_addr]
SSTORE                                            // []

PUSH20 0xC0FFEE0000000000000000000000000000000000 // [coffee_addr]
PUSH1 0x01                                        // [0x01, coffee_addr]
SSTORE                                            // []

// Storage:
// 0x00 -> deadbeefdeadbeefdeadbeefdeadbeefdeadbeef
// 0x01 -> c0ffee0000000000000000000000000000000000

PUSH1 0x00
SLOAD                                                                  // [dead_addr]
PUSH1 0x01
SLOAD                                                                  // [coffee_addr,
```

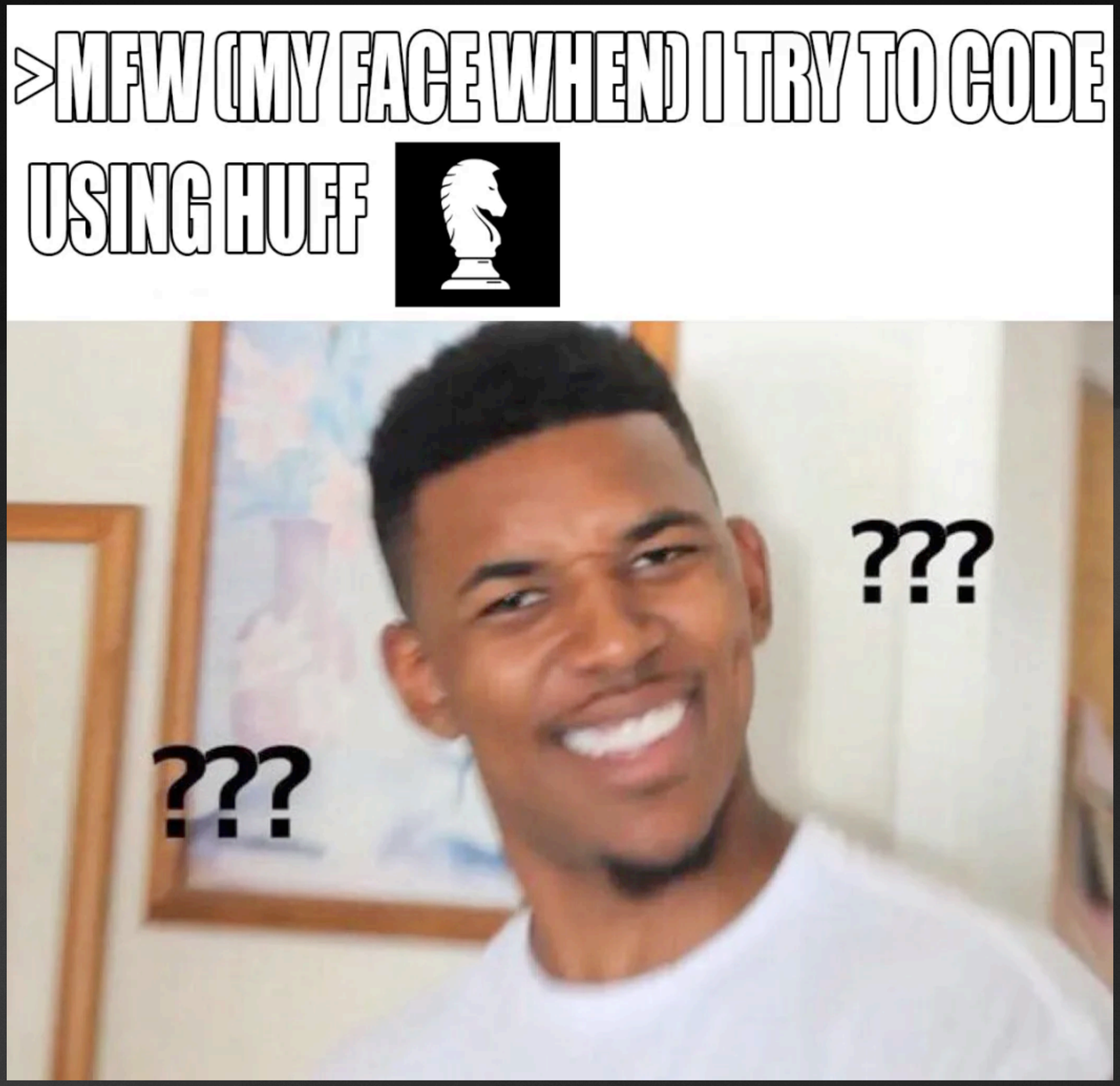
# Back to huff



# What the sigma

- Lets look at a basic huff contract:

```
#define function addTwo(uint256,uint256) view returns(uint256)

#define macro MAIN() = takes(0) returns(0) {

    // Get the function selector
    0x00
    calldataload
    0xE0
    shr

    // Jump to the implementation of the ADD_TWO function
    // if the calldata matches the function selector
    __FUNC_SIG(addTwo) eq addTwo jumpi

    addTwo:
        ADD_TWO()
}

#define macro ADD_TWO() = takes(0) returns(0) {
    0x04 calldataload     // load first 32 bytes onto the stack - number 1
    0x24 calldataload     // load second 32 bytes onto the stack - number 2
    add                   // add number 1 and 2 and put the result onto the stack

    0x00 mstore           // place the result in memory
    0x20 0x00 return      // return the result
}
```


>MFW (MY FACE WHEN) I TRY TO CODE
USING HUFF
???
???

# ABI declaration

- If you're familiar with higher-level languages like Solidity or Vyper, you know about defining "external" or "public" functions to interact with contracts externally via an ABI (Application Binary Interface).
- Huff works similarly, allowing you to declare functions that appear in the ABI at the top of the file.

```
#define function addTwo(uint256, uint256) view returns(uint256)
```

# Main!

- The MAIN directive is annotated with takes(0) returns(0), indicating it takes nothing from the stack and returns nothing, as the stack is empty when entering the contract and nothing is left on completion.

```
#define macro MAIN() = takes(0) returns(0) {
    0x00 calldataload     // [number1] // load first 32 bytes onto the stack - number 1
    0x20 calldataload     // [number2] // load second 32 bytes onto the stack - number 2
    add                   // [number1+number2] // add number 1 and 2 and put the result onto the stack

    0x00 mstore           // place [number1 + number2] in memory
    0x20 0x00 return      // return the result
}
```

# Modifying our contract to accept external function calls

- To accept external calls for multiple functions, extract the addTwo logic into another macro and convert the MAIN macro into a function dispatcher.
- Modify the ADD_TWO macro by shifting the calldata offset by 4 bytes for both numbers to account for the 4-byte function selector.
- The MAIN macro's first 4 lines isolate the function selector from calldata:
  - 0x00 pushes [0] onto the stack.
  - calldataload takes [0] as input and pushes the first 32 bytes of calldata onto the stack.
  - 0xE0 pushes [224] onto the stack, representing 256 bits - 32 bits (28 bytes).
  - shr shifts out calldata by 28 bytes, placing the function selector onto the stack.
- Subsequent lines match the function selector on the stack and jump to the appropriate code location, with Huff handling the jump logic. The ADD_TWO() macro bytecode is inlined in the main macro.

# We’re so back

```
#define function addTwo(uint256,uint256) view returns(uint256)

#define macro MAIN() = takes(0) returns(0) {

    // Get the function selector
    0x00
    calldataload
    0xE0
    shr

    // Jump to the implementation of the ADD_TWO function
    // if the calldata matches the function selector
    __FUNC_SIG(addTwo) eq addTwo jumpi

    addTwo:
        ADD_TWO()
}

#define macro ADD_TWO() = takes(0) returns(0) {
    0x04 calldataload     // load first 32 bytes onto the stack - number 1
    0x24 calldataload     // load second 32 bytes onto the stack - number 2
    add                   // add number 1 and 2 and put the result onto the stack

    0x00 mstore           // place the result in memory
    0x20 0x00 return      // return the result
}
```

# thanks;