1. Data types in Solidity:

* Value type
* Reference type

1. Value Type: It holds data within the memory owned by it (same as location of the variable). Max size allowed is 32bytes. When assigned to another variable or sent as an argument in a function, a new variable instance is created and the value is copied into it. Both the variables will be independent of each other.

* bool
* uint
* int
* address
* byte
* enum

1. Reference type: Values are stored separately. The variable only stores the address/pointer of the memory location. They can acquire value of more than 32bytes. When assigned to another variable or sent as an argument in another function, the pointer is copied to new variable rather than the value. Change in original or new variable will change the value in both the variables (as both point to the same memory location).

* arrays
* structs
* string
* mappings

1. Storage and Memory Data Location:

Variables can either have storage data location or memory data location, depending on its use. Storage variables are costly as it occupies permanent space on the EVM. Some comparative features are listed below:

|  |  |
| --- | --- |
| **Storage** | **Memory** |
| Global | Within Function |
| Permanent | Temporary |
| State Variable | Function Parameter |
| Expensive | Cheaper |

Data location depends on the following two factors:

* 1. Location of variable declaration
  2. Data type of variable (i.e., Value type or reference type)

1. Rules for data location of variables:
   1. Rule 1:

State variables, whether value type or reference type, are stored in storage data location. No need to mention storage or memory while declaring state variables.

* 1. Rule 2:

Variables declared as function parameters including return parameters are always stored in memory data location. Memory data location needs to be specified for reference data types to avoid compilation error. Not applicable for mapping as mapping cannot be declared as function parameter.

* 1. Rule 3: (Declarations within a function)

Value type variables declared within a function are always stored in memory (memory data location) and cannot be stored as storage.

Reference type variables require explicit specification while declaration as to whether the variable is to be stored in storage or memory. Furthermore, Reference type variables stored as storage should always point to a state variable therefore, value cannot be assigned directly and it only creates a reference to state variable (does not create a copy). Mapping cannot be declared in a function, however, mappings in a function can refer to mappings declared as state variables.

* 1. Rule 4:

Arguments supplied by callers to function parameters are always stored in calldata data location.

**Assignment Behaviour**

* 1. Rule 5:

Assignment to state variable from another state variable always creates a new copy whether it is a value type or reference type variable.

* 1. Rule 6:

Assignment of memory variable to state variable always creates a new copy whether it is a value type or reference type variable.

* 1. Rule 7:

Assignment of state variable to a memory variable always creates a new copy whether it is value type or reference type variable.

* 1. Rule 8:

Assignment from one memory variable to another memory variable:

* Creates a copy for value type variables
* Do not create a copy for reference type variables

1. Bytes:

Refers to 8bit signed integers. Everything is stored in bits consisting of binary values 0 and 1.

Ranges from bytes1 to bytes32. It cannot directly store integers. Need to convert to binary first.

1. Arrays:

Arrays can have a compile time fixed (pre-determined) size or they can be dynamic. Default value in fixed array is 0 for all indexes. Accessing non-existent element would result in run time error.

1. Address:

It can store 20bytes of data. It is specifically designed to hold account addresses in Ethereum (both for EOA and contract). It has a property ‘balance’ which returns balance available in that account in wei. While declaration if payable qualifier is used with address, it will provide the ability to receive and send ethers. It has 5 functions:

* Send (returns true / false)
* Transfer (similar to send, however incase of an error, reverts the transaction)
* Call
* Delegatecall
* Staticcall

1. Mapping:

Mapping represents a key value pair. Key is unique. Struct and enum can be used as a value but not as a key. It is similar to hash table and dictionary however, iteration is not possible here. Further, mapping can only be declared as a state variable with storage as memory location. Mapping can be declared in a function if they refer to mappings declared in state variable (Refer Rule3 above). Further, mapping cannot be returned from function. Values can also have further mappings i.e., mapping within a mapping (nested mapping).

Data type needs to defined at the time of declaration of variable and thereafter data type of that variable cannot be changed

1. Implicit conversion (data type):

Conversion will be automatic. It will happen where smaller data type is stored in a larger data type. In this case there will be no data loss and are completely type safe.

1. Explicit conversion:

This refers to situations where larger data type is converted into smaller data type. This type of conversion is not automatic and will require external help. Solidity provides a function for each value type for explicit conversion.

|  |
| --- |
| Syntax:  uint256 a = 10  uint16 b = uint16(a) |

Here data over and above the space available in the smaller data type will be lost.

1. Block/ Transaction Global Variable and Functions:

Solidity provides access to a few global variables that are not declared within contracts but are accessible from code within contracts.

|  |  |  |
| --- | --- | --- |
| **GLOBAL VARIABLE** | **DATA TYPE** | **DETAILS** |
| block.coinbase | address | Refer’s to miner’s address |
| block.difficulty | uint | Difficulty level of current block |
| block.gaslimit | uint | Gas limit for current block |
| block.number | uint | Current block’s number in sequence |
| block.timestamp | uint | Time when block was created |
| msg.data | bytes | Information about the function and its parameters that created the transaction |
| gasleft | uint | Gas unused after execution of transaction |
| msg.sender | address | address of the caller who invoked the function. It refers to the immediate account (it could be EOA or another contract account |
| msg.sig | bytes4 | Function identifier using first four bytes after hashing function signature. |
| msg.value | uint | Amount of wei sent along with transaction (works only in payable function) |
| now | uint | Current block time stamp |
| tx.gasprice | uint | The gas price caller is ready to pay for each gas unit |
| tx.origin | address | The first caller of the transaction (it will always be external account) |
| blockhash(uint blocknumber) returns(bytes32) |  | Hash of the block containing the transaction with a limitation of 256 recent blocks excluding current block |

1. Contract Global Variables:

A Solidity contract has a global variable ‘this’ and a global function self-destruct. ‘This’ is used to refer the current contract and it is explicitly convertible to address (address(this)). ’Selfdestruct’ function is used to destroy the current contract and send its funds to the given address.