



SOLIDITY
INTERMEDIATE

CONTENTS

Reference Data Type

Data Locations

Array

Structures

Mapping

Global Variables

Much more

BASIC DATA TYPE

Integer

Bool

Address

Byte

REFERENCE DATA TYPE

Strings

Arrays

Mappings

Struct

REFERENCE DATA TYPE

- Data types such as Array, Structs, Mapping are known as reference data type.
- If we use a reference type, we always have to explicitly provide the data area where the type is stored.
- Every reference type has an additional annotation, the “data location”, about where it is stored. There are three data locations: memory, storage and calldata.

DATA LOCATIONS

Code Eater

Data Location	About
Memory	lifetime is limited to an external function call
Storage	the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract
Calldata	special data location that contains the function arguments

ARRAYS

Code Eater

Fixed Array

Dynamic Array

FIXED ARRAY

Code Eater

- `uint[number_of_elements] public arr;`
- `uint[5] public arr;`

0	0	0	0	0
0	1	2	3	4

- `uint[5] public arr = [10,20,30,40,50];`

10	20	30	40	50
0	1	2	3	4



FIXED ARRAY

```
contract FixedSizeArray {  
    // Declare a fixed-size array of uint with length 5  
    uint[5] public fixedArray;  
  
    // Function to get the length of the fixed-size array  
    function getLength() public view returns (uint) {  
        return fixedArray.length;  
    }  
  
    // Function to update an element in the fixed-size array  
    function updateElement(uint index, uint value) public {  
        require(index < fixedArray.length, "Index out of bounds");  
        fixedArray[index] = value;  
    }  
}
```

[Click Here](#)



FIXED ARRAY EXAMPLE 2

[Click Here](#)

DYNAMIC ARRAY

Code Eater

- `uint[] public arr;`
- To insert an element we use `arr.push(element)`
- To remove an element we use `arr.pop()` . It removes the last element every time it is executed.
- To find the length of an array we use `arr.length`. It returns the length of the array in `uint` data type.



DYNAMIC ARRAY

```
contract DynamicSizeArray {  
    // Declare a dynamic-size array of uint  
    uint[] public dynamicArray;  
  
    // Function to add an element to the dynamic-size array  
    function addElement(uint value) public {    ⛊ 46873 gas  
        dynamicArray.push(value);  
    }  
  
    // Function to get the length of the dynamic-size array  
    function getLength() public view returns (uint) {    ⛊ 2467 gas  
        return dynamicArray.length;  
    }  
  
    // Function to remove the last element from the dynamic-size array  
    function removeLastElement() public {    ⛊ 31554 gas  
        require(dynamicArray.length > 0, "Array is already empty");  
        dynamicArray.pop();  
    }  
}
```

[Click Here](#)



STRING DATA TYPE

```
contract StringExample {  
    // State variable to store a string  
    string public myString;  
  
    // Function to set a new string  
    function setString(string memory newString) public {  
        myString = newString;  
    }  
  
    // Function to get the current string (optional, since we have a public variable)  
    function getString() public view returns (string memory) {  
        return myString;  
    }  
}
```

[Click Here](#)

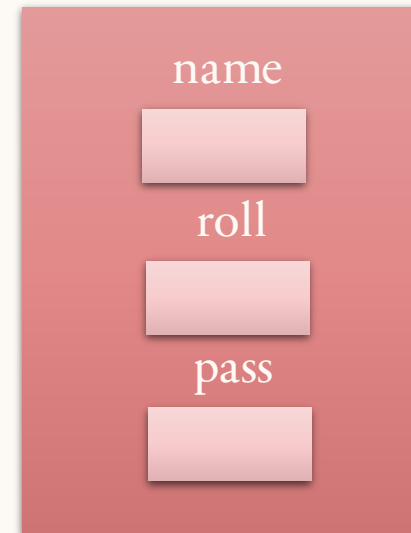
STRUCT

Code Eater

- Struct is a complex data type. A complex data type is **usually a composite of other existing data types**.

```
struct Student{  
    string name;  
    uint roll;  
    bool pass;  
}
```

- struct_type public var_name;
- **Student** public s1;





STRUCT

```
contract Classroom {  
    struct Student {  
        string name;  
        uint roll;  
        bool pass;  
    }  
  
    Student public s1;  
  
    function insert() public returns (Student memory) {  
        s1.name = "Kshitij";  
        s1.roll = 12;  
        s1.pass = true;  
        return s1;  
    }  
}
```

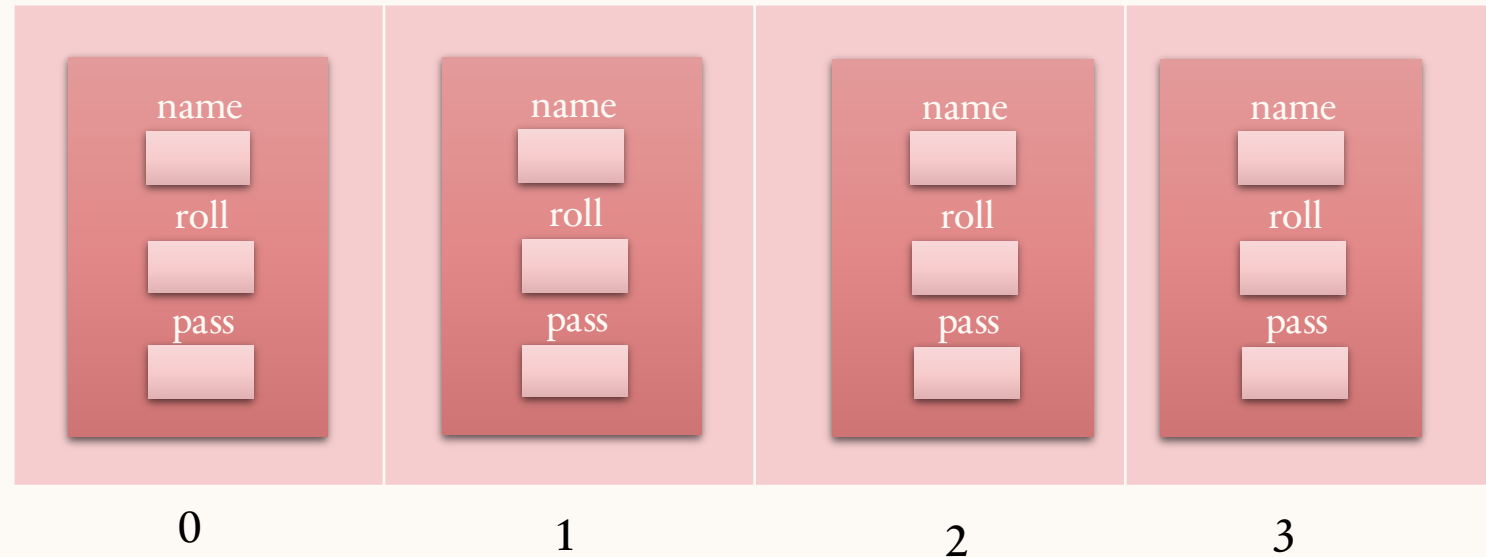
[Click Here](#)

ARRAY OF STRUCT

Code Eater

```
struct Student{  
    string name;  
    uint roll;  
    bool pass;  
}
```

Student[4] public s1;



MAPPING

Code Eater

- Concept of keys and values.
- mapping(key => value)
- mapping(uint => string) public data;

```
contract Grades {  
    // Define a mapping from student addresses to their grades  
    mapping(address => uint) public grades;  
  
    // Function to set the grade for a student  
    function setGrade(address student, uint grade) public { 2283  
        grades[student] = grade;  
    }  
  
    // Function to get the grade of a student  
    function getGrade(address student) public view returns (uint) {  
        return grades[student];  
    }  
}
```

[Click
Here](#)

SIMPLE MAPPING

Code Eater

_roll(uint)	name(string)

SIMPLE MAPPING

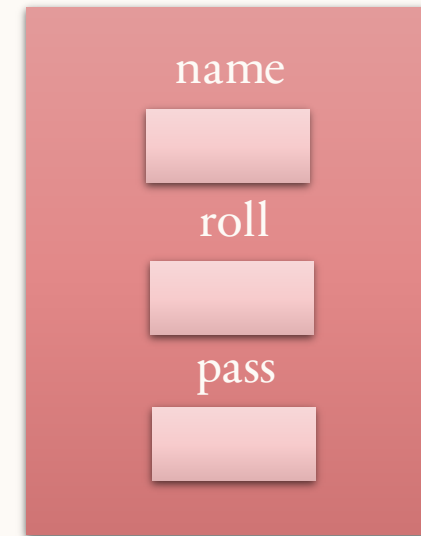
Code Eater

- mapping(uint => string) public data;

_roll		name
1	→	Ravi
5	→	John
40	→	Alice
50	→	Akash

MAPPING WITH STRUCT

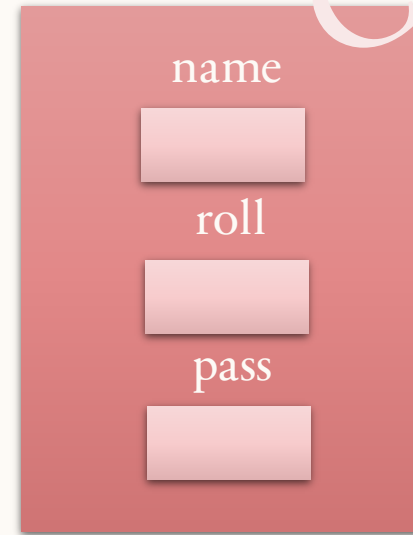
- `mapping(uint=>Student) public data;`



Student

- mapping(uint=>Student) public data;

uint	Student
0	<div><div>name</div><div></div><div>roll</div><div></div><div>pass</div><div></div></div>
12	<div><div>name</div><div></div><div>roll</div><div></div><div>pass</div><div></div></div>
100	<div><div>name</div><div></div><div>roll</div><div></div><div>pass</div><div></div></div>



Student

MAPPING WITH STRUCT

roll		_name	_class	_age
1	→	John	7	13
5	→	Alice	8	14
10	→	Ravi	3	9

MAPPING WITH ARRAY

mapping(address=>uint[]) private marks;

address	uint[]				
0xabc	10	20	30	40	50
0xdef	1	2	3		
0x121	121	221	243	52	
0x23f	12	22	23		
0x73e	10	20	30	40	50

NESTED MAPPING

Code Eater

```
mapping(address=>mapping(address=>bool)) private check;
```

	address1	address2	address3
address1		true	
address2			false
address3	false		

check[address1][address2] = true

check[address2][address3] = false

check[address3][address1] = false

NESTED MAPPING

Code Eater

```
mapping(uint=>mapping(uint=>bool)) private check;
```

0 1 2

0

1

2

	true	
		false
false		

check[0][1] = true

check[1][2] = false

check[2][0] = false

NESTED MAPPING

Code Eater

```
mapping(address=>mapping(address=>bool)) ownership;
```

	address1	address2	address3
address1		true	
address2			false
address3	false		

ownership[address1][address2] = true

ownership[address2][address3] = false

ownership[address3][address1] = false

GLOBAL VARIABLE

blockhash(uint blockNumber) returns (bytes32): hash of the given block when blocknumber is one of the 256 most recent blocks; otherwise returns zero

block.basefee (uint): current block's base fee ([EIP-3198](#) and [EIP-1559](#))

block.chainid (uint): current chain id

block.coinbase (address payable): current block miner's address

block.difficulty (uint): current block difficulty

block.gaslimit (uint): current block gaslimit

block.number (uint): current block number

block.timestamp (uint): current block timestamp as seconds since unix epoch

gasleft() returns (uint256): remaining gas

msg.data (bytes calldata): complete calldata

msg.sender (address): sender of the message (current call)

msg.sig (bytes4): first four bytes of the calldata (i.e. function identifier)

msg.value (uint): number of wei sent with the message

tx.gasprice (uint): gas price of the transaction

tx.origin (address): sender of the transaction (full call chain)

VOTING CONTRACT

Code Eater

Template - [Click Here](#)

Final Code(not yet) - [Click Here](#)

IMPORTANT POINTS

Code Eater

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract sample{
    address public owner;
    constructor(){
        owner=msg.sender;
    }
    function addressUser() public view returns(address){
        require(msg.sender==owner,"You are not the owner");
        return msg.sender;
    }
}
```

PAYABLE ADDRESS

Code Eater

- Same as `address`, but with the additional members `transfer` and `send`.

```
//      function contractTransfer() public payable{
//          //code or no code
//      }
//      function contractBalance() public view returns(uint){
//          return address(this).balance;
//      }

//      function sendUser(address payable reciever) public payable{
//          payable(reciever).transfer(msg.value);
//      }
// }
```

PAYABLE FUNCTION

Code Eater

- If a function is payable, this means that it also accepts a payment of zero Ether, so it also is non-payable.
- On the other hand, a non-payable function will reject Ether sent to it, so non-payable functions cannot be converted to payable functions.

DYNAMIC ARRAY – BYTE

Code Eater

- `bytes1[] public arr;` • `bytes2[] public arr;` • `bytes32[] public arr;`
- Array member functions are push, pop and length.

DYNAMIC ARRAY - BYTE

Code Eater

```
contract demo{
    bytes[] public arr1;

    function insertArray() public
    {
        arr1.push("abcde");
        arr1.push("b");
    }
}
```

Example - 7

DATA LOCATIONS

Code Eater

Data Location	About
Memory	lifetime is limited to an external function call
Storage	the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract
Calldata	special data location that contains the function arguments

DATA LOCATIONS

Code Eater

Data Location	About
Memory	lifetime is limited to an external function call
Storage	the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract
Calldata	special data location that contains the function arguments

MEMORY VS CALldata

Code Eater

Memory	Calldata
Can be used anywhere inside a function	Can be used only in a function argument
Mutable	Immutable

DATA LOCATIONS

Code Eater

```
contract demo{  
  
    function returnArray() public pure returns(uint[5] memory)  
    {  
        uint[5] memory arr;  
        return arr;  
    }  
}
```

Example - 1

DATA LOCATIONS

Code Eater

```
contract demo{
```

```
    function returnArray(uint[5] memory _arr,uint length) public pure returns(uint[5] memory)
    {
        for(uint i=0;i<length;i++){
            _arr[i]=_arr[i]*2;
        }
        return _arr;
    }
}
```

Example - 2

DATA LOCATIONS

Code Eater

```
contract demo{  
  
function returnArray(uint[5] calldata _arr,uint length) public pure returns(uint[5] memory)  
{  
    uint[5] memory brr;  
    for(uint i=0;i<length;i++){  
        brr[i]=_arr[i]*2;  
    }  
    return _arr;  
}  
}
```

Example - 3

DATA LOCATIONS

Code Eater

Data Location	About
Memory	lifetime is limited to an external function call
Storage	the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract
Calldata	special data location that contains the function arguments

DATA LOCATIONS

Code Eater

Data Location	About
Memory	lifetime is limited to an external function call
Storage	the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract
Calldata	special data location that contains the function arguments

MEMORY VS STORAGE

Code Eater

```
uint[3] public arr =[ 10,20,30];
```

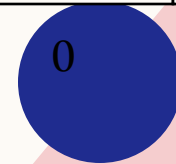
10	20	30
0	1	2

```
uint[3] memory arr1 = arr;
```

10	20	30
0	1	2

```
arr1[0] = 90;
```

arr1



MEMORY VS STORAGE

Code Eater

```
uint[3] public arr =[ 10,20,30];
```

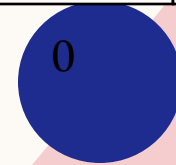
10	20	30
0	1	2

```
uint[3] memory arr1 = arr;
```

90	20	30
0	1	2

```
arr1[0] = 90;
```

arr1



MEMORY VS STORAGE

Code Eater

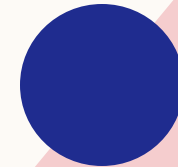
```
uint[3] public arr =[10,20,30];
```

10	20	30
0	1	2

```
uint[3] storage arr2 = arr;
```

```
arr2[0] = 90;
```

arr2



MEMORY VS STORAGE

Code Eater

```
uint[3] public arr =[ 10,20,30];
```

```
uint[3] storage arr2 = arr;
```

```
arr2[0] = 90;
```

arr2

90	20	30
0	1	2

STRING VS BYTES

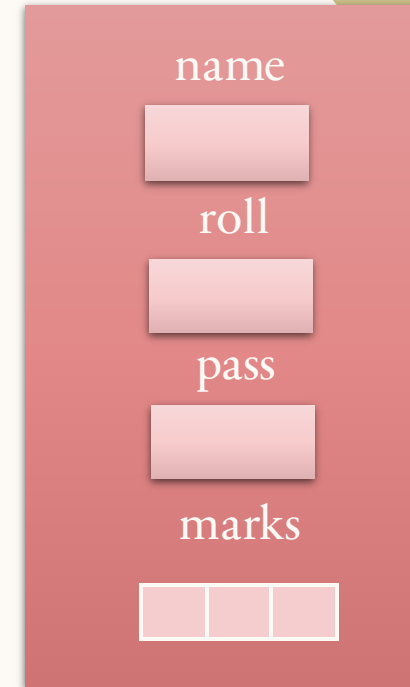
Code Eater

- Variables of type `bytes` and `string` are special arrays.
- `string` is equal to `bytes` but does not allow length or index access.
- By explicit type casting you can convert bytes form to string form.

ARRAY WITHIN STRUCT

Code Eater

```
struct Student{  
    string name;  
    uint roll;  
    bool pass;  
    uint8[3] marks;  
}
```



A diagram illustrating the memory layout of a C++ struct named 'Student'. The struct is represented as a vertical red rectangle. Inside, the fields are listed from top to bottom: 'name', 'roll', 'pass', and 'marks'. Each of the first three fields is followed by a single pink rectangular box representing its memory allocation. The 'marks' field is followed by a row of three adjacent pink rectangular boxes, representing an array of three elements. The background of the slide features large, overlapping triangles in shades of green and red.

name
roll
pass
marks

MAPPING VS ARRAY

Code Eater

_roll		name
1	→	Ravi
2	→	John
4	→	Alice
7	→	Akash

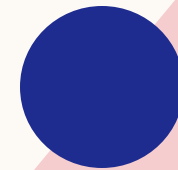
Mapping

MAPPING VS ARRAY

Code Eater

index	name
0	
1	Ravi
2	John
3	
4	Alice
5	
6	
7	Akash

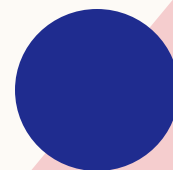
Array



MAPPING VS ARRAY

Code Eater

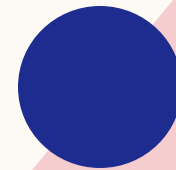
index	name
0	
1	Ravi
2	John
3	
4	Alice
5	
6	
7	Akash



MAPPING VS ARRAY

Code Eater

- Mapping stores data in non-contiguous fashion while array stores data in a contiguous fashion.
- Mappings are not iterable while arrays are iterable.

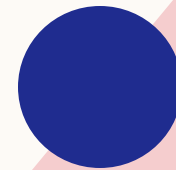


IMPORTANT POINTS

Code Eater

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.9.0;

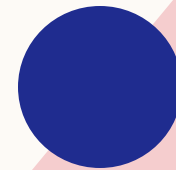
contract Demo {
    mapping(uint=>uint) public arr;
    function check() public {
        mapping(uint=>uint) storage arr1=arr;//referencing
        arr1[1]=2;
    }
}
```



IMPORTANT POINTS

Code Eater

- Mapping cannot be iterated.
- Mappings can only have a data location of storage and thus are allowed for state variables. In simple words, you cannot declare mapping inside a function.
- They cannot be used as parameters or return parameters of contract functions that are publicly visible.
- These restrictions are also true for arrays and structs that contain mappings.



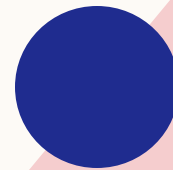
STRING DATA TYPE

Code Eater

- `string` public str ="hello world";
- Solidity does not have string manipulation functions, but there are third-party string libraries.
- We can compare two strings by their keccak256 hash using-
`keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))`
- Concatenate two strings using- `string.concat(s1,s2)`

QUIZ TIME - 3

Code Eater



Notes

FALLBACK

Code Eater

- The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function.
- Fallback function is optionally payable.
- A contract can have at most one fallback function, declared using either `fallback () external [payable]` or `fallback (bytes calldata input) external [payable] returns (bytes memory output)`
- This function must have external visibility.
- A fallback function can be virtual, can override and can have modifiers.
- The fallback function always receives data, but in order to also receive Ether it must be marked payable.

RECEIVE

Code Eater

- A contract can have at most one receive function, declared using `receive() external payable { ... }`.
- This function cannot have arguments, cannot return anything, and must have external visibility and payable state mutability.
- It can be virtual, can override and can have modifiers.
- If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.
- Using payable fallback functions for receiving Ether is not recommended.

What are data types?

Data types in programming languages are like different containers that can hold specific kinds of data. Just as you can put different things in a box or a bag, you can assign different types of values to different data types in programming. Here are some examples:

1. Integer: This data type is used to represent whole numbers. For example, you might use an integer data type to represent the number of items in a shopping cart, like this: ``num_items = 5``. Here, ``num_items`` is the name of the variable, and ``5`` is the integer value assigned to it.

2. String: This data type is used to represent text or characters. For example, you might use a string data type to represent a person's name, like this: ``name = "Alice"``. Here, ``name`` is the variable name, and ``"Alice"`` is the string value assigned to it.

3. Boolean: This data type is used to represent true or false values. For example, you might use a boolean data type to represent whether an item is in stock or not, like this: ``in_stock = True``. Here, ``in_stock`` is the variable name, and ``True`` is the boolean value assigned to it.

What are reference data types in solidity?

In Solidity, a reference data type is a type of data that stores a reference to a location in memory where the actual data is stored. This is different from a value data type, which stores the actual data directly in the variable.

There are three main reference data types in Solidity:

- 1. Arrays:** An array is a collection of values of the same data type. In Solidity, arrays can be either fixed-size or dynamic. A fixed-size array has a fixed length that is specified when the array is declared, while a dynamic array can grow or shrink in size as needed. When you declare an array variable, you are creating a reference to a location in memory where the array is stored.
- 2. Structs:** A struct is a user-defined data type that groups together related data of different data types. When you declare a struct variable, you are creating a reference to a location in memory where the struct is stored.
- 3. Mapping:** A mapping is a key-value data structure that allows you to store and retrieve values based on a key. When you declare a mapping variable, you are creating a reference to a location in memory where the mapping is stored.

What are reference data types in solidity?

Reference data types are useful in Solidity because they allow you to work with complex data structures and pass them as function arguments. However, they require more memory than value data types, and you need to be careful when working with them to avoid memory leaks or other issues.

Data Locations

In Solidity, there are several different data locations that determine where a variable's data is stored and how it can be accessed. These data locations include:

1. **Memory:** This is a temporary storage location used for variables that are only needed during the execution of a function. Memory variables are allocated dynamically and do not persist outside the function call.
2. **Storage:** This is a permanent storage location used for variables that need to be stored between function calls. Storage variables are persistent and are written to the blockchain.
3. **Calldata:** This is a read-only data location used for function arguments and return values. Calldata variables are used when you want to pass data into a function without modifying it.
4. **Stack:** This is a temporary storage location used for local variables and function arguments. Stack variables are allocated automatically when a function is called and are released when the function returns.

Arrays

In Solidity, an array is a data structure that allows you to store multiple values of the same data type in a contiguous block of memory. Solidity supports two types of arrays:

1. Fixed-size arrays: In a fixed-size array, the number of elements in the array is specified when the array is declared, and it cannot be changed afterward. For example, the following code declares a fixed-size array of 5 integers:

```
```
```

```
uint[5] myArray;
```

```
```
```

2. Dynamic arrays: In a dynamic array, the number of elements can be changed dynamically during runtime. Dynamic arrays are declared using the keyword `[]`. For example, the following code declares a dynamic array of integers:

```
```
```

```
uint[] myArray;
```

```
```
```

Arrays

Once an array is declared, you can access its elements using an index. The index of the first element is 0, and the index of the last element is the length of the array minus 1. For example, to access the third element of an array, you would use the index ``2``.

Solidity arrays also provide several built-in functions and properties that allow you to work with them more efficiently. For example, the ``length`` property returns the number of elements in an array, and the ``push()`` function adds an element to the end of a dynamic array.

Arrays in Solidity are commonly used for storing lists of data, such as addresses, integers, or structures. They can be used in a variety of use cases, such as storing historical data, managing a group of related values, or implementing data structures like stacks and queues.

Mapping

In Solidity, a mapping is a data structure that maps keys of one data type to values of another data type. It is like a dictionary or hash table in other programming languages. Each key in a mapping is unique, and the value associated with a key can be updated or read using the key.

A mapping is declared using the following syntax:

```
...  
mapping (KeyType => ValueType) myMapping;  
...
```

Here, `KeyType` is the data type of the keys in the mapping, and `ValueType` is the data type of the values associated with the keys. For example, the following code declares a mapping that maps addresses to integers:

```
...  
mapping (address => uint) balances;  
...
```

Mapping

Once a mapping is declared, you can add, update, or read values using the keys. For example, to add a value to a mapping, you would use the following syntax:

```
```\nmyMapping[key] = value;\n```
```

Here, `key` is the key you want to add or update, and `value` is the value associated with the key.

To read a value from a mapping, you would use the following syntax:

```
```\nValueType value = myMapping[key];\n```
```

Here, `key` is the key you want to read, and `ValueType` is the data type of the value associated with the key.

Mappings in Solidity are commonly used for storing and accessing large amounts of data quickly based on a unique key. They can be used in a variety of use cases, such as managing account balances, storing data about individual users, or implementing data structures like graphs and trees.

Arrays vs Mapping

In Solidity, an array is a data structure that allows you to store multiple values of the same data type in a contiguous block of memory. An array can be dynamically sized or fixed size, and its elements are accessed using an index.

On the other hand, a mapping is a data structure that maps keys of one data type to values of another data type. In other words, it is like a dictionary or hash table in other programming languages. Each key in a mapping is unique, and the value associated with a key can be updated or read using the key.

The key differences between an array and a mapping in Solidity are:

- 1. Indexing:** In an array, elements are accessed using an index, which is an integer value that represents the position of the element in the array. In a mapping, elements are accessed using a key, which can be of any non-array data type.
- 2. Size:** The size of an array is fixed at the time of creation or can be dynamic, but it cannot be changed once created. The size of a mapping is not fixed and can be changed dynamically by adding or removing key-value pairs.

Arrays vs Mapping

3. Initialization: In Solidity, arrays can be initialized with values when they are declared. However, mappings cannot be initialized with values at the time of declaration.

4. Iteration: It is possible to iterate over an array using a `for` loop or a `while` loop. In contrast, it is not possible to iterate over a mapping directly, but it is possible to iterate over the keys of a mapping and access the values using the keys.

In general, mappings are useful when you need to store a large number of key-value pairs and access them quickly based on their keys, while arrays are useful when you need to store and access a sequence of values.

String

In Solidity, ``string`` is a dynamic array of UTF-8 encoded characters, used to represent human-readable text.

Here's an example of how to declare a ``string`` variable:

```
...  
string memory greeting = "Hello, world!";  
...
```

In this example, ``greeting`` is a ``string`` variable that is assigned the value "Hello, world!".

You can also declare a ``string`` variable without initializing it:

```
...  
string memory myString;  
...
```

In this case, ``myString`` is a ``string`` variable that is initially empty.

String

Strings in Solidity support various operations such as concatenation, slicing, and length. Here are some examples:

- Concatenation: You can concatenate two strings using the ``abi.encodePacked`` function or the ``+`` operator.

```
...
```

```
string memory firstName = "John";  
string memory lastName = "Doe";  
string memory fullName = string(abi.encodePacked(firstName, " ", lastName)); // "John Doe"  
...
```

- Slicing: You can slice a string to get a substring using the ``bytes`` type and the ``abi.encodePacked`` function.

```
...
```

```
string memory message = "Hello, world!";  
bytes memory sliced = abi.encodePacked(bytes(message)[7:12]); // "world"  
...
```

String

In this example, ``bytes(message)[7:12]`` returns a slice of the ``message`` string from index 7 (inclusive) to index 12 (exclusive), which is the substring "world". The ``abi.encodePacked`` function is used to convert the slice to a ``bytes`` type.

- Length: You can get the length of a string using the ``bytes`` type.

```
```\nstring memory message = "Hello, world!";\nuint256 length = bytes(message).length; // 13\n```
```

In this example, ``bytes(message).length`` returns the length of the ``message`` string in bytes, which is 13.

It's important to note that operations on ``string`` variables can be expensive in terms of gas costs, especially when dealing with large strings. It's generally a good practice to minimize the use of ``string`` variables in smart contracts and use more efficient data types like ``bytes`` whenever possible.

# Bytes

In Solidity, ``bytes`` is a dynamic array of bytes, used to represent arbitrary binary data. It is similar to ``string``, but can also be used to represent non-textual data such as images, audio, or video.

Here's an example of how to declare a ``bytes`` variable:

```
```\nbytes memory myBytes = new bytes(10);\n```
```

In this example, ``myBytes`` is a ``bytes`` variable that is initialized with a length of 10 bytes. You can also initialize a ``bytes`` variable with a value:

Bytes Vs String

In Solidity, ``bytes`` and ``string`` are two different data types used to represent sequence of characters.

``bytes`` is an array of bytes, where each byte is a number from 0 to 255, and it is used to represent arbitrary binary data. ``bytes`` is a dynamic type, meaning that its length can be changed at runtime.

``string`` is an array of UTF-8 encoded characters, and it is used to represent human-readable text. ``string`` is also a dynamic type and its length can be changed at runtime.

Here's an example of how to declare a ``bytes`` variable:

```
...  
bytes memory data = new bytes(10);  
...
```

This creates a ``bytes`` variable called ``data`` with a length of 10 bytes.

Bytes Vs String

Here's an example of how to declare a `string` variable:

```
```\nstring memory message = "Hello, world!";\n```
```

This creates a `string` variable called `message` with a value of "Hello, world!".

While both `bytes` and `string` can represent sequences of characters, there are some differences between them:

- `bytes` can be used to represent any kind of binary data, while `string` is designed specifically for human-readable text.
- `bytes` can be more efficient in terms of gas costs when working with non-textual data, while `string` can be more efficient when working with textual data.
- `bytes` can be used with low-level byte manipulation operations like bitwise operations, while `string` cannot.
- `bytes` can be converted to `string`, and vice versa, using conversion functions like `string(bytes memory)` and `bytes(string memory)`, but there is some overhead associated with these conversions.

In general, if you are working with arbitrary binary data, use `bytes`. If you are working with human-readable text, use `string`.

# Enum

In Solidity, ``enum`` is a user-defined value type used to represent a set of named values. It is similar to an enumerated type in other programming languages.

An ``enum`` declaration defines a new enumerated type with a set of named values, where each value is represented by an integer constant. Here's an example of how to declare an ``enum``:

```
```\nenum Color {Red, Green, Blue}\n```
```

In this example, ``Color`` is the name of the new enumerated type, and ``Red``, ``Green``, and ``Blue`` are the named values of the ``enum``. By default, the integer values of the named values start at 0 and increase by 1 for each subsequent value. In this case, ``Red`` is 0, ``Green`` is 1, and ``Blue`` is 2.

Enum

Once an `enum` is defined, you can create variables of that type and assign them one of the named values. Here's an example of how to declare a variable of type `Color`:

```
```\nColor myColor = Color.Red;\n```
```

In this example, `myColor` is a variable of type `Color` that is assigned the value `Red`.

You can also use `enums` in function arguments and return values, and in conditional statements like `if` and `switch`. Here's an example of how to use an `enum` in a conditional statement:

# Enum

```
...
function getColorName(Color color) public pure returns (string memory) {
 if (color == Color.Red) {
 return "Red";
 } else if (color == Color.Green) {
 return "Green";
 } else {
 return "Blue";
 }
}
...
```

In this example, the function `getColorName` takes a `Color` argument and returns a string representing the name of the color. The function uses a conditional statement to check the value of the `Color` argument and returns the corresponding name.

`enums` can be useful for defining a set of related constants and improving code readability.

# Struct

In Solidity, a struct is a custom data type that allows you to define a collection of related variables with different data types. It is similar to a struct in other programming languages like C and C++.

A struct is declared using the following syntax:

```
...
struct Person {
 string name;
 uint age;
 bool isStudent;
}
...
```

Here, `Person` is the name of the struct, and `name`, `age`, and `isStudent` are variables of different data types.

# Struct

Once a struct is declared, you can create new instances of the struct and access its variables. For example, to create a new instance of the `Person` struct, you would use the following syntax:

```
```\nPerson myPerson = Person("John", 25, true);\n```
```

Here, `myPerson` is an instance of the `Person` struct with the name "John", age 25, and is a student.

To access the variables in a struct, you would use the dot notation. For example, to access the age variable in the `myPerson` instance, you would use the following syntax:

```
```\nuint myAge = myPerson.age;\n```
```

Here, `myAge` is assigned the value 25, which is the age of the `myPerson` instance.

# Global Variables

In Solidity, global variables are variables that can be accessed from any part of a contract without having to pass them as parameters. They are predefined variables that hold information about the current state of the contract, the blockchain, and the environment in which the contract is running.

Here are some of the most commonly used global variables in Solidity:

1. ``msg.sender``: This variable holds the address of the account that called the current function.
2. ``msg.value``: This variable holds the amount of Ether that was sent along with the current function call.
3. ``block.timestamp``: This variable holds the timestamp of the current block in seconds since the Unix epoch.
4. ``block.number``: This variable holds the number of the current block in the blockchain.
5. ``address(this)``: This variable holds the address of the current contract.
6. ``now``: This variable is an alias for ``block.timestamp`` and holds the current timestamp in seconds since the Unix epoch.



# Global Variables

Global variables can be used to implement certain functionalities in smart contracts, such as time-based operations, contract-to-contract interactions, and access control. They can be accessed from any part of the contract, including functions, events, and modifiers.

It's important to note that global variables can also be manipulated by attackers or malicious users, so it's important to use them with caution and implement proper security measures to prevent vulnerabilities in the contract.

# Delete Keyword

In Solidity, the `delete` keyword is used to set a variable or a mapping element to its default value. It is often used to reset the state of a variable or a mapping element to its initial state, or to remove a mapping element altogether.

Here are some examples of how the `delete` keyword can be used:

1. Resetting a variable to its default value:

```
...
uint myNumber = 10;
delete myNumber; // sets myNumber to 0
...
```

2. Removing an element from a mapping:

```
...
mapping (address => uint) balances;
balances[msg.sender] = 100;
delete balances[msg.sender]; // removes the element at msg.sender key
...
```

# Delete Keyword

3. Resetting a struct to its default value:

```
```\n\nstruct Person {\n    string name;\n    uint age;\n}\n\nPerson myPerson = Person("John", 25);\ndelete myPerson; // sets myPerson to an empty Person struct with default values\n```\n
```

It's important to note that the `delete` keyword is not available for arrays in Solidity. To remove an element from an array, you can set the value of the element to its default value or use other array manipulation techniques such as shifting elements.

It's also important to be careful when using the `delete` keyword, as it can have unintended consequences if used improperly. For example, deleting a mapping element that doesn't exist can lead to an out-of-gas error, and deleting a complex data structure like a struct or an array can have unexpected results if not done correctly.

Payable Address

In Solidity, a payable address is an Ethereum address that can receive Ether payments. It is a type of address that can be used to send Ether to a contract or an account on the Ethereum network.

When a function in a contract is marked as ``payable``, it means that it can receive Ether payments from a payable address. For example, the following function is marked as payable:

```
...  
function receivePayment() public payable {  
    // do something with the payment  
}  
...
```

Here, the ``payable`` keyword indicates that this function can receive Ether payments from a payable address. When someone calls this function and sends Ether along with the call, the amount of Ether sent will be stored in the ``msg.value`` global variable.

Payable Address

Payable addresses can also be used to transfer Ether between accounts or contracts. For example, the following code transfers 1 Ether from the contract to the address specified:

```
```\naddress payable receiver = 0x1234567890123456789012345678901234567890;\nreceiver.transfer(1 ether);\n```
```

Here, `receiver` is a payable address, and the `transfer` function is used to send 1 Ether to that address. The `payable` keyword is used to indicate that the address can receive Ether payments.

It's important to note that sending Ether to a non-payable address will result in an exception and the transaction will fail. Additionally, sending Ether to a contract that doesn't have a payable function will result in the same exception. Therefore, it's important to ensure that the target address is a payable address before sending any Ether to it.

# Payable Function

In Solidity, a payable function is a function that can receive Ether along with its execution. It is marked with the `payable` modifier, which allows the function to receive Ether payments from a sender.

Here's an example of a payable function in Solidity:

```
...
function receivePayment() public payable {
 // do something with the payment
}
...
```

In this example, the `payable` keyword indicates that this function can receive Ether payments. When someone calls this function and sends Ether along with the call, the amount of Ether sent will be stored in the `msg.value` global variable, and the function can perform some actions with the payment received.

It's important to note that a payable function can only receive Ether payments if it is called from a payable address. A payable address is an Ethereum address that can send and receive Ether payments. To send Ether to a payable function, you need to call it from a payable address and specify the amount of Ether to be sent in the transaction.

# Payable Function

Here's an example of how to call a payable function and send Ether to it:

```
...
```

```
address payable receiver = 0x1234567890123456789012345678901234567890;
receiver.receivePayment.value(1 ether)();
...
```

In this example, `receiver` is a payable address, and the `receivePayment` function is a payable function that can receive Ether payments. The `value` keyword is used to specify the amount of Ether to be sent (in this case, 1 ether), and the `()` at the end of the function call indicates that no arguments are being passed to the function.

It's important to handle Ether payments in a secure and robust manner to prevent vulnerabilities in the contract. Always validate and sanitize input data, implement proper access controls, and handle exceptions and errors gracefully.

[www.codeeater.in](http://www.codeeater.in)

**THANK YOU**

Code Eater

Instagram - @codeeater21

LinkedIn – @kshitijWeb3