LAB 02 August, 2023

1 Structs vs Enums

Structs and enums are both powerful features in programming languages like Solidity, but they serve different purposes.

1) **Purpose:** Structs allow you to define custom data types with multiple properties or fields, making them useful for grouping related data and representing complex entities. Enums, on the other hand, restrict a variable to a predefined set of values, providing a way to represent and enforce a finite set of options or choices.

2) **Usage**: Structs are commonly used for modeling records or entities, such as representing a student with properties like name, age, and grade. Enums, on the other hand, are often used to define a set of named values, like representing the different states of an order (e.g., "pending," "processing," "completed").

3) **Composition:** Structs can contain multiple variables of different data types, allowing for the creation of more complex data structures. Enums, on the other hand, are typically composed of a single variable restricted to a limited set of named values.

In summary, while structs enable the creation of custom data types with multiple properties, enums provide a means to restrict variables to a predefined set of named values.

They serve different purposes and are used in different contexts within programming languages like Solidity.

EXERCISE: "Ticket Booking System"

Design and implement a Solidity contract named **TicketBookingSystem** for a simple ticket booking system with the following features:

1) enum '**TicketType'**: Representing the ticket types available for booking, namely Economy, Business, and FirstClass.

```
enum TicketType { ... }
```

2) enum '**Days'**: Representing the days of the week on which flights are available, including Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.

```
enum Days {...}
```

3) struct '**Ticket'**: Storing the details of each booked ticket, including the 'passenger name', 'ticketType', 'day', and 'isBooked' status (a boolean indicating if the ticket is available or booked).

```
struct Ticket {
        string passenger;
        TicketType ticketType;
        Days day;
        bool isBooked;
    }
```

4) **Ticket[]** public tickets: A dynamic array to store booked tickets' information.

```
Ticket[] public tickets;
```

5) function **bookTicket** : Allows users to book a ticket with the given passenger name, day, and desired ticketType. The booked ticket is added to the tickets array.

```
function bookTicket(string memory _passenger, Days _day, TicketType
_ticketType) public
```

6) function **getTicketType** : Takes the index of a booked ticket and returns its TicketType.

```
function getTicketType(uint256 _index) public view returns
(TicketType)
```

7) function **getTicketDay** : Takes the index of a booked ticket and returns the Days value representing the day of the week for the booked ticket.

```
 function getTicketDay(uint256 _index) public view returns (Days)
```

8) function **toggleStatus** : Takes the index of a booked ticket and toggles its isBooked status. If the ticket is available, it marks it as booked, and vice versa.

```
 function toggleStatus(uint256 _index) public
```

2 Conditional Loops

Loops in Solidity are fundamental programming constructs that allow for repetitive execution of code. They are useful when you need to perform a task multiple times or iterate over a collection of data. Solidity supports the following types of loops:

2.1 While Loop:

- The while loop is a basic loop that repeatedly executes a block of code as long as a specified condition is true.
- It is commonly used when the number of iterations is unknown beforehand.

Syntax:

```
while (condition) {
    // Code to be executed
}
```

The loop first checks the condition. If it evaluates to true, the code inside the loop is executed.

After each iteration, the condition is checked again until it becomes false, at which point the loop terminates

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract Loop {
 function PrimeChecker(uint _num) public pure returns (bool is_prime ) {
   uint j =2;
   is_prime= true;
   if (_num == 0 || _num == 1) {
       is_prime = false;
   }
   while (j <= _num/2 ) {
      if (_num % j == 0) {
    is_prime = false;
    break;  }
    j++;      }
}
}
```

2.2 Do-While Loop:

- The do-while loop is similar to the while loop, but it guarantees that the code block is executed at least once, even if the condition is initially false.
- The condition is checked at the end of the loop.

Syntax:

```
do {
    // Code to be executed
} while (condition);
```

The code block is executed first, and then the condition is evaluated. If the condition is true, the loop continues to the next iteration. If the condition is false, the loop terminates.

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract Loop {
   function Palindrome(uint input) public pure returns (bool) {
    uint temp = input;
    uint digit;
    uint rev;
  do{
         digit = temp % 10;
         rev = (rev * 10) + digit;
         temp = temp / 10;
     } while (temp != 0);

     if (input == rev)
         return true;
     else
          return false;                                          }
}
```

A **palindrome number** is a number that is same after reverse. For example 121, 34543, 343, 131, 48984 are the palindrome numbers.

2.3 For Loop:

- The for loop is a compact and commonly used loop when the number of iterations is known in advance.
- It consists of three parts: initialization, condition, and iteration statement.
- The loop continues executing as long as the condition is true.

Syntax:

```
for (initialization; condition; iteration statement) {
    // Code to be executed
}
```

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Loop {

 uint[] data;
 function loop() public returns (uint[] memory) {

    for (uint8 i = 0; i < 20; i=i+2) {
        data.push(i);
    }
    return data;
}
}
```

EXERCISE:

1. Deploy and observe the above smart contract.
2. Modify above smart contract such that it returns **odd** numbers from **0 - 20.**

Hint: Odd numbers aren't divisible by 2
e.g 1,3,5,7……..

2.4 Practical Usage

- ✓ Loops in Solidity are essential for iterating over arrays, processing lists, performing calculations, and more.
- ✓ However, it's crucial to ensure that loops have proper termination conditions to avoid infinite looping, which can consume excessive gas or result in unexpected behavior.
- ✓ Careful consideration should be given to the complexity and efficiency of the loop logic to optimize gas usage and contract perform

EXERCISE:

1. Deploy, observe and explain the following code.
2. Instead of **while** loop, use **for** loop and then **do while** loop, separately.
3. Try to return other variants of the **unit** and observe what happens.
4. Explain the difference between while and do while loop.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract ControlFlow {
 // while
    function whileTest() public pure returns(uint256){
        uint sum = 0;
        uint i = 0;
        while(i < 15){
            sum += i;
            i++;       }
        return(sum);
    }
}
```