# 7. 기본자료형, mapping, array, struct 타입의 상태 변수가 각각 storage에 어떻게 저장되는지 설명해주세요.

## 1) 타입 기본 설명

### (1) mapping

#### 타입 설명

`mapping` 타입은 두 개의 타입을 각각 `key`와 `value`로 맵핑하여 저장하는 변수이다. `hash table`과 비슷한 개념으로서 해당 `key`의 타입에서 사용가능한 모든 `key`에 대해 대응되는 `value`는 값을 할당하기 전에는 전부 초기 default 값인 0으로 맵핑된다. `mapping`은 다음과 같이 선언된다.

```
mapping(_KeyType => _ValueType)
```

그리고 맵핑의 `key` 데이터는 `mapping`에 저장되지 않으며, 오직 `keccak256` 해쉬값만 사용하여 `value`를 찾는다. 따라서 `mapping`은 길이를 갖지 않으며, `key`에 대한 정확한 값을 알지 못하면 해당 `value` 값을 삭제할 수 없다.

또한 `mapping`은 오직 `storage` 공간만 사용할 수 있기 때문에 `state variable`로만 사용될 수 있다.

#### 사용 예시

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

function update(uint newBalance) public {
    balances[msg.sender] = newBalance;
}
```

### (2) array

#### 타입 설명

`array` 타입은 컴파일 타임에 고정된 길이를 갖도록 선언될 수도 있고, 또는 동적인 크기를 갖을 수도 있다. 어떤 타입 T에 대하여 k개의 요소를 갖는 경우 `T[k]`로 표현한다. 이 때 타입 T 역시 `array` 타입이 될 수 있다. 만약 타입 T가 어떤 타입 P를 n개의 요소를 갖는 `array` 라면, `T[k] = P[n][k]` 가 된다.

그리고 만약 `array`가 동적인 크기를 갖는 경우 타입 T에 대한 동적인 크기의 `array`는 `T[]`로 표현한다. 그리고 만약 동적인 크기를 갖는 `array`가 모여서 또 다른 `array`를 이룰 경우 타입 T에 대한 동적 `array`를 k개 갖는 `array`는 `T[][k]`로 표현한다.

***`array`의 변수에 접근할 때는 선언할 때와 순서가 반대가 된다.*** 따라서 만약 `uint[][3] foo`에서 세 번째 동적 `array`에 있는 두 번째 `uint`에 접근할 때는 `foo[2][1]`로 접근한다.

> ***참고사항*** *:*
> *`bytes`와 `string`는 특별한 형태의 `array`이다. `bytes`는 `byte[]`와 비슷하지만 `bytes`가 메모리를 적게 소모하므로 더 효율적이다. `string`은 `bytes`와 비슷하지만 `length`와 `index` 접근을 허용하지 않는다.*

#### 사용 예시

```
bool[2][] pairsOfFlags;

pairsOfFlags[0][0] = true;
pairsOfFlags[0][1] = false;
pairsOfFlags[1][0] = true;
pairsOfFlags[1][1] = false;
pairsOfFlags[2][0] = false;
pairsOfFlags[2][1] = true;

uint[][4] scoreList;

scoreList[3][0] = 98;
scoreList[2][5] = 67;
scoreList[1][12] = 100;
scoreList[0][7] = 86;
```

### (3) struct

#### 타입 설명

`struct` 타입은 구조체 형식으로 여러 멤버들을 정의하여 새로운 타입을 만들 수 있는 방법이다.

> ***참고사항*** *:*
> *`struct` 타입은 내부 변수의 타입에 자기 자신인 `struct` 타입을 지정할 수 없다. 하지만 `mapping`의 `value` 타입으로 선언되거나 또는 동적 크기를 갖는 자기 자신의 `array` 타입은 가질 수 있다. `storage` 영역에 저장된 `struct` 타입의 변수를 로컬 변수에 할당하면, 해당 `struct`의 멤버들이 복사되는 것이 아니라 reference를 저장하게 된다. 따라서 이 경우 로컬 변수 값을 변경하면 state 값도 변경된다.*

#### 사용 예시

```
pragma solidity >=0.4.11 <0.7.0;

// Defines a new type with two fields.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
  // Structs can also be defined inside contracts, which makes them
  // visible only there and in derived contracts.
  struct Campaign {
    address payable beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
  }
```

```
  uint numCampaigns;
  mapping (uint => Campaign) campaigns;

  function newCampaign(address payable beneficiary, uint goal) public returns (uint
campaignID) {
    campaignID = numCampaigns++;
    campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
  }

  function contribute(uint campaignID) public payable {
    Campaign storage c = campaigns[campaignID];
    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
    c.amount += msg.value;
  }
}
```

## 2) 타입별 storage 저장 구조

### 1) Storage 설명

각 스마트 컨트랙트는 자신만의 `persistent` 한 `storage` 를 갖고 있다. 그리고 모든 `state variables` 는 이 `storage` 에 저장된다.

이 `storage` 는 하나의 거대한 `document` 라고 생각할 수 있다. 이 때 `key` 들은 32바이트의 스트링이며 총 $2^{256}$개의 `key` 를 가질 수 있다. `value` 도 마찬가지로 32바이트 크기를 갖으며, 총 $2^{256}$개의 `key` 에 대응되는 `value` 를 가질 수 있다. 그리고 할당되지 않은 `key` 에 대한 `value` 는 모두 0으로 초기화되어 있다.

이것은 간단하게 생각하면 $2^{256}$ 개의 element를 갖는 매우 거대한 `array` 라고 볼 수 있으며, `EVM` 에서는 각 `key` 를 인덱스로 갖는 여러개의 `slot` 으로 표현한다. 따라서 인덱스 0부터 시작하여 slot[0], slot[1], slot[2]..., slot[$2^{256}$-1]까지의 `slot` 들의 저장공간이 있는 것이다.

| slot | 0 | 1 | 2 | 3 | 4 | 5 | ... | ... | $2^{256}$-1 |
|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 0 | 0 | 0 | 0 | 0 | ... | ... | 0 |

> ***참고사항*** *:*
> *`value` 에 0을 저장하는 것은 공간을 차지하지 않기 때문에 `storage` 는 `value` 에 0을 할당함으로써 사용하던 저장 공간을 반납할 수 있다. 이럴 경우 트랜잭션 처리시 보상으로 일정량의 gas를 refund 받게 된다.*

### 2) Statically-sized 변수

고정된 길이를 갖는 변수들은 선언된 순서대로 `storage` 의 `slot` 들에 순차적으로 저장된다. 따라서 변수를 아래와 같이 선언하면 `slot` 에는 선언된 순서대로 값들이 저장된다.

```
uint a;
uint256 b;
uint8 c;
```

```
uint8 d;
uint16 e;
```

| slot | 0 | 1 | 2 | ... |
|---|---|---|---|---|
| variable | a | b | c \| d \| e | ... |

`struct` 타입을 선언할 때도 내부의 멤버 변수들의 순서에 따라 `slot` 에 저장된다.

```
struct Group {
  uint8 a;
  uint8 b;
  uint16 c;
  uint256 d;
  int e;
}

Group myGroup;
```

| slot | 0 | 1 | 2 | ... |
|---|---|---|---|---|
| variable | a \| b \| c | d | e | ... |

고정된 길이의 `array` 타입을 선언할 때도 순차적으로 `slot` 에 저장된다.

```
uint128[2] a;
uint256[2] b;
```

| slot | 0 | 1 | 2 | ... |
|---|---|---|---|---|
| variable | a[0] \| a[1] | b[0] | b[1] | ... |

#### Tight Variable Packing

`storage` 에 변수가 저장될 때 한 `slot` 의 크기는 32바이트이다. 만약 32바이트 보다 작은 변수들이 연속으로 선언되면 한 `slot` 안에 여러개의 변수가 저장될 수 있다. 하지만 크기를 합쳤을 때 32바이트가 넘어가면 해당 변수는 다음 `slot` 에 저장되어야 한다.

```
struct Group {
  uint8 a;
  uint256 b;
  uint8 c;
  uint16 d;
  int e;
}
```

```
Group myGroup;
```

| slot | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| variable | a | b | c \| d | e | ... |

그리고 `struct` 타입과 `array` 타입이 선언될 때는 항상 새로운 `slot` 부터 시작하여 순차적으로 저장된다. 아래의 예제를 보면 k 변수 이후에 해당 `slot` 에는 많은 공간 여유가 있지만 `struct` 타입의 시작이므로 다음의 `slot[3]` 부터 저장된다.

```
struct Group {
  uint8 a;
  uint256 b;
  uint8 c;
  uint16 d;
  int e;
}

uint8 m;
int n;
uint16 k;
Group myGroup;
uint256[2] f;
```

| slot | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| variable | m | n | k | a | b | c \| d | e | f[0] | f[1] | ... |

위 예제에서 본 것처럼 크기가 작은 변수들을 연속으로 함께 선언할 수록 `slot` 에 저장되는 공간을 줄일 수 있고 `storage` 를 `read` , `write` 할 때도 한 번에 많은 변수를 읽고 쓸 수 있으므로 연산수가 줄어들어 효율적이다. 이것은 `struct` 내에서 멤버들을 선언할 때도 마찬가지로 적용된다. 따라서 이렇게 최대한 `slot` 공간을 효율적으로 활용하기 위해 변수들의 선언 순서를 조정하는 것을 `Tight Variable Packing` 패턴이라고 한다. 위의 예제에서 `Tight Variable Packing` 패턴을 적용하면 다음과 같이 `slot` 사용이 줄어든다.

```
struct Group {
  uint8 a;
  uint8 c;
  uint16 d;
  uint256 b;
  int e;
}

uint8 m;
uint16 k;
int n;
```

```
Group myGroup;
uint256[2] f;
```

| slot | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| variable | m \| k | n | a \| c \| d | b | e | f[0] | f[1] | ... |

### 3) Dynamically-sized array

동적의 크기를 갖는 `array` 는 정적인 크기를 갖는 변수들과는 다른 방식으로 저장된다. 동적인 `array` 변수를 저장할 때는 해당 타입의 크기와 데이터가 저장된다.

동적인 크기를 갖는 `array` 는 크기를 미리 알 수 없기 때문에 `slot` 인덱스를 임의로 결정해야 하는데, 전체 `slot` 의 개수가 $2^{256}$개로 매우 크기 때문에 인덱스로 해쉬값을 사용한다. 이 때 해당 `array` element의 길이가 저장되는 `slot` 의 인덱스의 해쉬값을 계산하여 value가 저장되는 `slot` 의 인덱스로 사용한다. `array` 의 value가 저장되는 인덱스는 다음과 같이 계산된다.

```
function arrLocation(uint256 slot, uint256 index, uint256 elementSize) public pure
returns (uint256)
{
  return uint256(keccak256(slot)) + (index * elementSize);
}
```

동적 크기를 갖는 `array` c는 다음과 같이 `slot` 에 저장된다.

```
uint a;
uint b;
uint256[] c;
```

| slot | 0 | 1 | 2 | ... | hash(2) | hash(2)+1 | ... |
|---|---|---|---|---|---|---|---|
| variable | a | b | c.length | ... | c[0] | c[1] | ... |

### 4) Mapping

`mapping` 에도 몇 개의 `key-value` 가 할당될 지 미리 알 수 없기 때문에 해쉬를 사용하여 `slot` 을 인덱싱한다. `mapping` 변수의 어떤 `key` 에 대한 `value` 값을 갖고 있는 `slot` 의 인덱스를 계산할 때는 다음과 같이 해당 `key` 와 `mapping` 변수가 선언된 `slot` 인덱스를 이용해 해쉬값을 계산한다.

```
function mapLocation(uint256 slot, uint256 key) public pure returns (uint256) {
    return uint256(keccak256(key, slot));
}
```

`mapping` 의 value들은 다음과 같이 `slot` 에 저장된다. 즉 `array` 와는 다르게 연속적이지 않고 `key` 마다 계산된 해쉬값에 따라 완전히 동떨어진 곳에 각 `value` 가 분산되어 저장된다. 그리고 이 때 해당 `mapping` 변수 위치의 `slot` 인덱스에는 아무런 값도 저장되지 않는다.

```
uint a;
uint b;
mapping(uint => uint) public c;
```

| slot | 0 | 1 | 2 | ... | hash(78, 2) | ... | hash(312, 2) | ... |
|---|---|---|---|---|---|---|---|---|
| variable | a | b | | ... | c[78] | ... | c[312] | ... |

---

**References**

[1] *Steve Marx. (2018, Mar 9).* *Understanding Ethereum Smart Contract Storage* *[Program The Blockchain]*