Storing Key-Value Pairs with Mappings

Storage mappings in Cairo provide a way to associate keys with values and persist them in the contract's storage. Unlike traditional hash tables, storage mappings do not store the key data itself; instead, they use the hash of the key to compute an address that corresponds to the storage slot where the corresponding value is stored. Therefore, it is not possible to iterate over the keys of a storage mapping.

    {{#label fig-mappings}}
    Figure {{#ref fig-mappings}}: Mapping keys to values in storage

Mappings do not have a concept of length or whether a key-value pair is set. All values are by default set to 0. As such, the only way to remove an entry from a mapping is to set its value to the default value for the type, which would be `0` for the `u64` type.

The `Map` type, provided by the Cairo core library, inside the `core::starknet::storage` module, is used to declare mappings in contracts.

To declare a mapping, use the `Map` type enclosed in angle brackets `<>`, specifying the key and value types. In Listing {{#ref storage-mappings}}, we create a simple contract that stores values mapped to the caller's address.

> The `Felt252Dict` type is a **memory** type that cannot be stored in contract storage. For persistent storage of key-value pairs, use the `Map` type, which is a [phantom type] [phantom types] designed specifically for contract storage. However, `Map` has limitations: it can't be instantiated as a regular variable, used as a function parameter, or included as a member in regular structs. `Map` can only be used as a storage variable within a contract's storage struct. To work with the contents of a `Map` in memory or perform complex operations, you'll need to copy its elements to and from a `Felt252Dict` or other suitable data structure.

Declaring and Using Storage Mappings

<!-- TODO PHANTOM TYPES -->
<!-- [phantom types]: ./ch11-03-intro-to-phantom-data.html -->

```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_02_storage_mapping/src/lib.cairo:contract}}
```

{{#label storage-mappings}}
Listing {{#ref storage-mappings}}: Declaring a storage mapping in the Storage struct

To read the value corresponding to a key in a mapping, you first need to retrieve the storage pointer associated with that key. This is done by calling the `entry` method on the storage mapping variable, passing in the key as a parameter. Once you have the entry path, you can call the `read` function on it to retrieve the stored value.

```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
```

listing_02_storage_mapping/src/lib.cairo:read}}
```
Similarly, to write a value in a storage mapping, you need to retrieve the storage pointer corresponding to the key. Once you have this storage pointer, you can call the `write` function on it with the value to write.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_02_storage_mapping/src/lib.cairo:write}}
```
Nested Mappings
You can also create more complex mappings with multiple keys. To illustrate this, we'll implement a contract representing warehouses assigned to users, where each user can store multiple items with their respective quantities.
The `user_warehouse` mapping is a storage mapping that maps `ContractAddress` to another mapping that maps `u64` (item ID) to `u64` (quantity). This can be implemented by declaring a `Map<ContractAddress, Map<u64, u64>>` in the storage struct. Each `ContractAddress` key in the `user_warehouse` mapping corresponds to a user's warehouse, and each user's warehouse contains a mapping of item IDs to their respective quantities.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_nested_storage_mapping/src/lib.cairo:storage}}
```
In this case, the same principle applies for accessing the stored values. You need to traverse the keys step by step, using the `entry` method to get the storage path to the next key in the sequence, and finally calling `read` or `write` on the innermost mapping.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_nested_storage_mapping/src/lib.cairo:accesses}}
```
Storage Address Computation for Mappings
The address in storage of a variable stored in a mapping is computed according to the following rules:
- For a single key `k`, the address of the value at key `k` is `h(sn_keccak(variable_name), k)`, where `h` is the Pedersen hash and the final value is taken modulo \\({2^{251}} - 256\\).
- For multiple keys, the address is computed as `h(...h(h(sn_keccak(variable_name), k_1), k_2), ..., k_n)`, with `k_1, ..., k_n` being all keys that constitute the mapping.
If the key of a mapping is a struct, each element of the struct constitutes a key. Moreover, the struct should implement the `Hash` trait, which can be derived with the `#[derive(Hash)]` attribute.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_02_storage_mapping/src/lib.cairo:struct_key_mapping}}
```
Summary
```

- Storage mappings allow you to map keys to values in contract storage.
- Use the `Map` type to declare mappings.
- Access mappings using the `entry` method and `read`/`write` functions.
- Mappings can contain other mappings, creating nested storage mappings.
- The address of a mapping variable is computed using the `sn_keccak` and the Pedersen hash functions.