Storing Collections with Vectors
The `Vec` type provides a way to store collections of values in the contract's storage. In this section, we will explore how to declare, add elements to and retrieve elements from a `Vec`, as well as how the storage addresses for `Vec` variables are computed.
The `Vec` type is provided by the Cairo core library, inside the `core::starknet::storage` module. Its associated methods are defined in the `VecTrait` and `MutableVecTrait` traits that you will also need to import for read and write operations on the `Vec` type.
> The `Array<T>` type is a **memory** type and cannot be directly stored in contract storage. For storage, use the `Vec<T>` type, which is a [phantom type][phantom types] designed specifically for contract storage. However, `Vec<T>` has limitations: it can't be instantiated as a regular variable, used as a function parameter, or included as a member in regular structs. To work with the full contents of a `Vec<T>`, you'll need to copy its elements to and from a memory `Array<T>`.
[phantom types]: ./ch11-05-phantom-data.html#phantom-type-in-generics
Declaring and Using Storage Vectors
To declare a Storage Vector, use the `Vec` type enclosed in angle brackets `<>`, specifying the type of elements it will store. In Listing {{#ref storage-vecs}}, we create a simple contract that registers all the addresses that call it and stores them in a `Vec`. We can then retrieve the `n`-th registered address, or all registered addresses.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_storage_vecs/src/lib.cairo:contract}}
```
{{#label storage-vecs}}
Listing {{#ref storage-vecs}}: Declaring a storage `Vec` in the Storage struct
To add an element to a `Vec`, you use the `append` method to get a storage pointer to the next available slot, and then call the `write` function on it with the value to add.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_storage_vecs/src/lib.cairo:append}}
```
To retrieve an element, you can use the `at` or `get` methods to get a storage pointer to the element at the specified index, and then call the `read` method to get the value. If the index is out of bounds, the `at` method panics, while the `get` method returns `None`.
```cairo, noplayground
{{#rustdoc_include ../listings/ch14-building-starknet-smart-contracts/
listing_storage_vecs/src/lib.cairo:read}}
```
If you want to retrieve all the elements of the Vec, you can iterate over the indices of the storage `Vec`, read the value at each index, and append it to a memory `Array<T>`. Similarly, you can't store an `Array<T>` in storage: you would need to iterate over the elements of the array and append them to a storage `Vec<T>`.
At this point, you should be familiar with the concept of storage pointers and storage paths introduced in the ["Contract Storage"][contract-storage] section and how they are

used to access storage variables through a pointer-based model. Thus how would you modify the address stored at a specific index of a `Vec`?

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

The answer is fairly simple: get a mutable pointer to the storage pointer at the desired index, and use the `write` method to modify the value at that index.

[contract-storage]: ./ch14-01-00-contract-storage.md

Storage Address Computation for Vecs

The address in storage of a variable stored in a `Vec` is computed according to the following rules:

- The length of the `Vec` is stored at the base address, computed as `sn_keccak(variable_name)`.
- The elements of the `Vec` are stored in addresses computed as `h(base_address, i)`, where `i` is the index of the element in the `Vec` and `h` is the Pedersen hash function.

Summary

- Use the `Vec` type to store collections of values in contract storage
- Access Vecs using the `append` method to add elements, and the `at` or `get` methods to read elements
- The address of a `Vec` variable is computed using the `sn_keccak` and the Pedersen hash functions

This wraps up our tour of the Contract Storage! In the next section, we'll start looking at the different kind of functions defined in a contract. You already know most of them, as we used them in the previous chapters, but we'll explain them in more detail.