Hashes

At its essence, hashing is a process of converting input data (often called a message) of any length into a fixed-size value, typically referred to as a "hash." This transformation is deterministic, meaning that the same input will always produce the same hash value. Hash functions are a fundamental component in various fields, including data storage, cryptography and data integrity verification. They are very often used when developing smart contracts, especially when working with [Merkle trees][merkle tree wiki].

In this chapter, we will present the two hash functions implemented natively in the Cairo core library: `Poseidon` and `Pedersen`. We will discuss when and how to use them, and see examples with Cairo programs.

[merkle tree wiki]: https://en.wikipedia.org/wiki/Merkle_tree#Uses

Hash Functions in Cairo

The Cairo core library provides two hash functions: Pedersen and Poseidon.

Pedersen hash functions are cryptographic algorithms that rely on [elliptic curve cryptography][ec wiki]. These functions perform operations on points along an elliptic curve — essentially, doing math with the locations of these points — which are easy to do in one direction and hard to undo. This one-way difficulty is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is a problem so hard to solve that it ensures the security of the hash function. The difficulty of reversing these operations is what makes the Pedersen hash function secure and reliable for cryptographic purposes.

Poseidon is a family of hash functions designed to be very efficient as algebraic circuits. Its design is particularly efficient for Zero-Knowledge proof systems, including STARKs (so, Cairo). Poseidon uses a method called a 'sponge construction,' which soaks up data and transforms it securely using a process known as the Hades permutation. Cairo's version of Poseidon is based on a three-element state permutation with [specific parameters][poseidon parameters].

[ec wiki]: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

[poseidon parameters]: https://github.com/starkware-industries/poseidon/blob/main/poseidon3.txt

When to Use Them?

Pedersen was the first hash function used on Starknet, and is still used to compute the addresses of variables in storage (for example, `LegacyMap` uses Pedersen to hash the keys of a storage mapping on Starknet). However, as Poseidon is cheaper and faster than Pedersen when working with STARK proofs system, it's now the recommended hash function to use in Cairo programs.

Working with Hashes

The core library makes it easy to work with hashes. The `Hash` trait is implemented for all types that can be converted to `felt252`, including `felt252` itself. For more complex types like structs, deriving `Hash` allows them to be hashed easily using the hash function of your choice - given that all of the struct's fields are themselves hashable. You cannot derive the `Hash` trait on a struct that contains un-hashable values, such as `Array<T>` or `Felt252Dict<T>`, even if `T` itself is hashable.

The `Hash` trait is accompanied by the `HashStateTrait` and `HashStateExTrait` that define the basic methods to work with hashes. They allow you to initialize a hash state that will contain the temporary values of the hash after each application of the hash

function, update the hash state and finalize it when the computation is completed.
`HashStateTrait` and `HashStateExTrait` are defined as follows:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_03_hash_trait/src/
lib.cairo:hashtrait}}
```
To use hashes in your code, you must first import the relevant traits and functions. In the following example, we will demonstrate how to hash a struct using both the Pedersen and Poseidon hash functions.
The first step is to initialize the hash with either `PoseidonTrait::new() -> HashState` or `PedersenTrait::new(base: felt252) -> HashState` depending on which hash function we want to work with. Then the hash state can be updated with the `update(self: HashState, value: felt252) -> HashState` or `update_with(self: S, value: T) -> S` functions as many times as required. Then the function `finalize(self: HashState) -> felt252` is called on the hash state and it returns the value of the hash as a `felt252`.
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_04_hash_poseidon/src/
lib.cairo}}
```
Pedersen is different from Poseidon, as it starts with a base state. This base state must be of `felt252` type, which forces us to either hash the struct with an arbitrary base state using the `update_with` method, or serialize the struct into an array to loop through all of its fields and hash its elements together.
Here is a short example of Pedersen hashing:
```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_04_hash_pedersen/src/
lib.cairo:main}}
```
Advanced Hashing: Hashing Arrays with Poseidon
Let us look at an example of hashing a struct that contains a `Span<felt252>`.
To hash a `Span<felt252>` or a struct that contains a `Span<felt252>` you can use the built-in function `poseidon_hash_span(mut span: Span<felt252>) -> felt252`. Similarly, you can hash `Array<felt252>` by calling `poseidon_hash_span` on its span.
First, let us import the following traits and function:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_05_advanced_hash/src/
lib.cairo:import}}
```
Now we define the struct. As you might have noticed, we didn't derive the `Hash` trait. If you attempt to derive the `Hash` trait for this struct, it will result in an error because the structure contains a field that is not hashable.
```cairo, noplayground
{{#include ../listings/ch11-advanced-features/no_listing_05_advanced_hash/src/
lib.cairo:structure}}
```
In this example, we initialized a `HashState` (`hash`), updated it and then called the

function `finalize()` on the
`HashState` to get the computed hash `hash_felt252`. We used `poseidon_hash_span`
on the `Span` of the `Array<felt252>` to compute its hash.
```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_05_advanced_hash/
src/lib.cairo:main}}
```