Dictionaries
Cairo provides in its core library a dictionary-like type. The `Felt252Dict<T>` data type represents a collection of key-value pairs where each key is unique and associated with a corresponding value. This type of data structure is known differently across different programming languages such as maps, hash tables, associative arrays and many others.

The `Felt252Dict<T>` type is useful when you want to organize your data in a certain way for which using an `Array<T>` and indexing doesn't suffice. Cairo dictionaries also allow the programmer to easily simulate the existence of mutable memory when there is none.

Basic Use of Dictionaries
It is normal in other languages when creating a new dictionary to define the data types of both key and value. In Cairo, the key type is restricted to `felt252`, leaving only the possibility to specify the value data type, represented by `T` in `Felt252Dict<T>`.

The core functionality of a `Felt252Dict<T>` is implemented in the trait `Felt252DictTrait` which includes all basic operations. Among them we can find:

1. `insert(felt252, T) -> ()` to write values to a dictionary instance and
2. `get(felt252) -> T` to read values from it.

These functions allow us to manipulate dictionaries like in any other language. In the following example, we create a dictionary to represent a mapping between individuals and their balance:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_09_intro/src/lib.cairo}}
```
We can create a new instance of `Felt252Dict<u64>` by using the `default` method of the `Default` trait and add two individuals, each one with their own balance, using the `insert` method. Finally, we check the balance of our users with the `get` method. These methods are defined in the `Felt252DictTrait` trait in the core library.

Throughout the book we have talked about how Cairo's memory is immutable, meaning you can only write to a memory cell once but the `Felt252Dict<T>` type represents a way to overcome this obstacle. We will explain how this is implemented later on in ["Dictionaries Underneath"][dict underneath].

Building upon our previous example, let us show a code example where the balance of the same user changes:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_10_intro_rewrite/src/lib.cairo}}
```
Notice how in this example we added the 'Alex' individual twice, each time using a different balance and each time that we checked for its balance it had the last value inserted! `Felt252Dict<T>` effectively allows us to "rewrite" the stored value for any given key.

Before heading on and explaining how dictionaries are implemented it is worth mentioning that once you instantiate a `Felt252Dict<T>`, behind the scenes all keys have their associated values initialized as zero. This means that if for example, you tried to get the balance of an inexistent user you will get 0 instead of an error or an undefined value. This also means there is no way to delete data from a dictionary.

Something to take into account when incorporating this structure into your code.
Until this point, we have seen all the basic features of `Felt252Dict<T>` and how it mimics the same behavior as the corresponding data structures in any other language, that is, externally of course. Cairo is at its core a non-deterministic Turing-complete programming language, very different from any other popular language in existence, which as a consequence means that dictionaries are implemented very differently as well!
In the following sections, we are going to give some insights about `Felt252Dict<T>` inner mechanisms and the compromises that were taken to make them work. After that, we are going to take a look at how to use dictionaries with other data structures as well as use the `entry` method as another way to interact with them.
[dict underneath]: ./ch03-02-dictionaries.md#dictionaries-underneath
Dictionaries Underneath
One of the constraints of Cairo's non-deterministic design is that its memory system is immutable, so in order to simulate mutability, the language implements `Felt252Dict<T>` as a list of entries. Each of the entries represents a time when a dictionary was accessed for reading/updating/writing purposes. An entry has three fields:
1. A `key` field that identifies the key for this key-value pair of the dictionary.
2. A `previous_value` field that indicates which previous value was held at `key`.
3. A `new_value` field that indicates the new value that is held at `key`.
If we try implementing `Felt252Dict<T>` using high-level structures we would internally define it as `Array<Entry<T>>` where each `Entry<T>` has information about what key-value pair it represents and the previous and new values it holds. The definition of `Entry<T>` would be:
```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_11_entries/src/lib.cairo:struct}}
```
For each time we interact with a `Felt252Dict<T>`, a new `Entry<T>` will be registered:
- A `get` would register an entry where there is no change in state, and previous and new values are stored with the same value.
- An `insert` would register a new `Entry<T>` where the `new_value` would be the element being inserted, and the `previous_value` the last element inserted before this. In case it is the first entry for a certain key, then the previous value will be zero.
The use of this entry list shows how there isn't any rewriting, just the creation of new memory cells per `Felt252Dict<T>` interaction. Let's show an example of this using the `balances` dictionary from the previous section and inserting the users 'Alex' and 'Maria':
```cairo
{{#rustdoc_include ../listings/ch03-common-collections/no_listing_11_entries/src/lib.cairo:inserts}}
```
These instructions would then produce the following list of entries:

key	previous	new
Alex	0	100

Maria	0	50
Alex	100	200
Maria	50	50

Notice that since 'Alex' was inserted twice, it appears twice and the `previous` and `current` values are set properly. Also reading from 'Maria' registered an entry with no change from previous to current values.

This approach to implementing `Felt252Dict<T>` means that for each read/write operation, there is a scan for the whole entry list in search of the last entry with the same `key`. Once the entry has been found, its `new_value` is extracted and used on the new entry to be added as the `previous_value`. This means that interacting with `Felt252Dict<T>` has a worst-case time complexity of `O(n)` where `n` is the number of entries in the list.

If you pour some thought into alternate ways of implementing `Felt252Dict<T>` you'd surely find them, probably even ditching completely the need for a `previous_value` field, nonetheless, since Cairo is not your normal language this won't work.

One of the purposes of Cairo is, with the STARK proof system, to generate proofs of computational integrity. This means that you need to verify that program execution is correct and inside the boundaries of Cairo restrictions. One of those boundary checks consists of "dictionary squashing" and that requires information on both previous and new values for every entry.

Squashing Dictionaries

To verify that the proof generated by a Cairo program execution that used a `Felt252Dict<T>` is correct, we need to check that there wasn't any illegal tampering with the dictionary. This is done through a method called `squash_dict` that reviews each entry of the entry list and checks that access to the dictionary remains coherent throughout the execution.

The process of squashing is as follows: given all entries with certain key `k`, taken in the same order as they were inserted, verify that the ith entry `new_value` is equal to the ith + 1 entry `previous_value`.

For example, given the following entry list:

key	previous	new
Alex	0	150
Maria	0	100
Charles	0	70
Maria	100	250
Alex	150	40
Alex	40	300
Maria	250	190
Alex	300	90

After squashing, the entry list would be reduced to:

key	previous	new
Alex	0	90
Maria	0	190
Charles	0	70

In case of a change on any of the values of the first table, squashing would have failed during runtime.
Dictionary Destruction
If you run the examples from ["Basic Use of Dictionaries"][basic dictionaries] section, you'd notice that there was never a call to squash dictionary, but the program compiled successfully nonetheless. What happened behind the scene was that squash was called automatically via the `Felt252Dict<T>` implementation of the `Destruct<T>` trait. This call occurred just before the `balance` dictionary went out of scope.
The `Destruct<T>` trait represents another way of removing instances out of scope apart from `Drop<T>`. The main difference between these two is that `Drop<T>` is treated as a no-op operation, meaning it does not generate new CASM while `Destruct<T>` does not have this restriction. The only type which actively uses the `Destruct<T>` trait is `Felt252Dict<T>`, for every other type `Destruct<T>` and `Drop<T>` are synonyms. You can read more about these traits in [Drop and Destruct][drop destruct] section of Appendix C.
Later in ["Dictionaries as Struct Members"][dictionaries in structs] section, we will have a hands-on example where we implement the `Destruct<T>` trait for a custom type.
[basic dictionaries]: ./ch03-02-dictionaries.md#basic-use-of-dictionaries
[drop destruct]: ./appendix-03-derivable-traits.md#drop-and-destruct
[dictionaries in structs]: ./ch11-01-custom-data-structures.md#dictionaries-as-struct-members
More Dictionaries
Up to this point, we have given a comprehensive overview of the functionality of `Felt252Dict<T>` as well as how and why it is implemented in a certain way. If you haven't understood all of it, don't worry because in this section we will have some more examples using dictionaries.
We will start by explaining the `entry` method which is part of a dictionary basic functionality included in `Felt252DictTrait<T>` which we didn't mention at the beginning. Soon after, we will see examples of how to use `Felt252Dict<T>` with other [complex types][nullable dictionaries values] such as `Array<T>`.
[nullable dictionaries values]: ./ch03-02-dictionaries.md#dictionaries-of-types-not-supported-natively
Entry and Finalize
In the ["Dictionaries Underneath"][dict underneath] section, we explained how `Felt252Dict<T>` internally worked. It was a list of entries for each time the dictionary was accessed in any manner. It would first find the last entry given a certain `key` and then update it accordingly to whatever operation it was executing. The Cairo language gives us the tools to replicate this ourselves through the `entry` and `finalize` methods.
The `entry` method comes as part of `Felt252DictTrait<T>` with the purpose of creating a new entry given a certain key. Once called, this method takes ownership of the dictionary and returns the entry to update. The method signature is as follows:
```cairo,noplayground
fn entry(self: Felt252Dict<T>, key: felt252) -> (Felt252DictEntry<T>, T) nopanic
```
The first input parameter takes ownership of the dictionary while the second one is used to create the appropriate entry. It returns a tuple containing a

`Felt252DictEntry<T>`, which is the type used by Cairo to represent dictionary entries, and a `T` representing the value held previously.
The `nopanic` notation simply indicates that the function is guaranteed to never panic.
The next thing to do is to update the entry with the new value. For this, we use the `finalize` method which inserts the entry and returns ownership of the dictionary:
```cairo,noplayground
fn finalize(self: Felt252DictEntry<T>, new_value: T) -> Felt252Dict<T>
```
This method receives the entry and the new value as parameters, and returns the updated dictionary.
Let us see an example using `entry` and `finalize`. Imagine we would like to implement our own version of the `get` method from a dictionary. We should then do the following:
1. Create the new entry to add using the `entry` method.
2. Insert back the entry where the `new_value` equals the `previous_value`.
3. Return the value.
Implementing our custom get would look like this:
```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_12_custom_methods/src/lib.cairo:imports}}
{{#include ../listings/ch03-common-collections/no_listing_12_custom_methods/src/lib.cairo:custom_get}}
```
The `ref` keyword means that the ownership of the variable will be given back at the end of
the function. This concept will be explained in more detail in the ["References and Snapshots"][references] section.
Implementing the `insert` method would follow a similar workflow, except for inserting a new value when finalizing. If we were to implement it, it would look like the following:
```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_12_custom_methods/src/lib.cairo:imports}}
{{#include ../listings/ch03-common-collections/no_listing_12_custom_methods/src/lib.cairo:custom_insert}}
```
As a finalizing note, these two methods are implemented in a similar way to how `insert` and `get` are implemented for `Felt252Dict<T>`. This code shows some example usage:
```cairo
{{#rustdoc_include ../listings/ch03-common-collections/no_listing_12_custom_methods/src/lib.cairo:main}}
```
[dict underneath]: ./ch03-02-dictionaries.md#dictionaries-underneath
[references]: ./ch04-02-references-and-snapshots.md
Dictionaries of Types not Supported Natively
One restriction of `Felt252Dict<T>` that we haven't talked about is the trait `Felt252DictValue<T>`.

This trait defines the `zero_default` method which is the one that gets called when a value does not exist in the dictionary.

This is implemented by some common data types, such as most unsigned integers, `bool` and `felt252` - but it is not implemented for more complex types such as arrays, structs (including `u256`), and other types from the core library.

This means that making a dictionary of types not natively supported is not a straightforward task, because you would need to write a couple of trait implementations in order to make the data type a valid dictionary value type.

To compensate this, you can wrap your type inside a `Nullable<T>`.

`Nullable<T>` is a smart pointer type that can either point to a value or be `null` in the absence of value. It is usually used in Object Oriented Programming Languages when a reference doesn't point anywhere. The difference with `Option` is that the wrapped value is stored inside a `Box<T>` data type. The `Box<T>` type is a smart pointer that allows us to use a dedicated `boxed_segment` memory segment for our data, and access this segment using a pointer that can only be manipulated in one place at a time. See [Smart Pointers Chapter](./ch11-02-smart-pointers.md) for more information.

Let's show using an example. We will try to store a `Span<felt252>` inside a dictionary. For that, we will use `Nullable<T>` and `Box<T>`. Also, we are storing a `Span<T>` and not an `Array<T>` because the latter does not implement the `Copy<T>` trait which is required for reading from a dictionary.

```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_13_dict_of_complex/src/
lib.cairo:imports}}
{{#include ../listings/ch03-common-collections/no_listing_13_dict_of_complex/src/
lib.cairo:header}}
//...
```

In this code snippet, the first thing we did was to create a new dictionary `d`. We want it to hold a `Nullable<Span>`. After that, we created an array and filled it with values.

The last step is inserting the array as a span inside the dictionary. Notice that we do this using the `new` function of the `NullableTrait`.

Once the element is inside the dictionary, and we want to get it, we follow the same steps but in reverse order. The following code shows how to achieve that:

```cairo,noplayground
//...
{{#include ../listings/ch03-common-collections/no_listing_13_dict_of_complex/src/
lib.cairo:footer}}
```

Here we:

1. Read the value using `get`.
2. Verified it is non-null using the `match_nullable` function.
3. Unwrapped the value inside the box and asserted it was correct.

The complete script would look like this:

```cairo
{{#include ../listings/ch03-common-collections/no_listing_13_dict_of_complex/src/
lib.cairo:all}}
```

~~~
## Using Arrays inside Dictionaries
In the previous section, we explored how to store and retrieve complex types inside a dictionary using `Nullable<T>` and `Box<T>`. Now, let's take a look at how to store an array inside a dictionary and dynamically modify its contents.
Storing arrays in dictionaries in Cairo is slightly different from storing other types. This is because arrays are more complex data structures that require special handling to avoid issues with memory copying and references.
First, let's look at how to create a dictionary and insert an array into it. This process is pretty straightforward and follows a similar pattern to inserting other types of data:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_14_dict_of_array_insert/src/lib.cairo}}
~~~
However, attempting to read an array from the dictionary using the `get` method will result in a compiler error. This is because `get` tries to copy the array in memory, which is not possible for arrays (as we've already mentioned in the [previous section][nullable dictionaries values], `Array<T>` does not implement the `Copy<T>` trait):
```cairo
{{#include ../listings/ch03-common-collections/no_listing_15_dict_of_array_attempt_get/src/lib.cairo}}
```
```shell
{{#include ../listings/ch03-common-collections/no_listing_15_dict_of_array_attempt_get/output.txt}}
```
To correctly read an array from the dictionary, we need to use dictionary entries. This allows us to get a reference to the array value without copying it:
```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_16_dict_of_array/src/lib.cairo:get}}
```
> Note: We must convert the array to a `Span` before finalizing the entry, because calling `NullableTrait::new(arr)` moves the array, thus making it impossible to return it from the function.
To modify the stored array, such as appending a new value, we can use a similar approach. The following `append_value` function demonstrates this:
```cairo,noplayground
{{#include ../listings/ch03-common-collections/no_listing_16_dict_of_array/src/lib.cairo:append}}
```
In the `append_value` function, we access the dictionary entry, dereference the array, append the new value, and finalize the entry with the updated array.
> Note: Removing an item from a stored array can be implemented in a similar manner.
Below is the complete example demonstrating the creation, insertion, reading, and modification of an array in a dictionary:

```cairo
{{#include ../listings/ch03-common-collections/no_listing_16_dict_of_array/src/lib.cairo:all}}
```
{{#quiz ../quizzes/ch03-02-dictionaries.toml}}