Custom Data Structures

When you first start programming in Cairo, you'll likely want to use arrays (`Array<T>`) to store collections of data. However, you will quickly realize that arrays have one big limitation - the data stored in them is immutable. Once you append a value to an array, you can't modify it.

This can be frustrating when you want to use a mutable data structure. For example, say you're making a game where the players have a level, and they can level up. You might try to store the level of the players in an array:

```cairo,noplayground
{{#include ../listings/ch11-advanced-features/listing_01_array_collection/src/lib.cairo:array_append}}
```

But then you realize you can't increase the level at a specific index once it's set. If a player dies, you cannot remove it from the array unless he happens to be in the first position.

Fortunately, Cairo provides a handy built-in [dictionary type](./ch03-02-dictionaries.md) called `Felt252Dict<T>` that allows us to simulate the behavior of mutable data structures. Let's first explore how to create a struct that contains, among others, a `Felt252Dict<T>`.

> Note: Several concepts used in this chapter were already presented earlier in the book. We recommend checking out the following chapters if you need to revise them:
> [Structs](ch05-00-using-structs-to-structure-related-data.md),
> [Methods](./ch05-03-method-syntax.md),
> [Generic types](./ch08-00-generic-types-and-traits.md),
> [Traits](./ch08-02-traits-in-cairo.md).

Dictionaries as Struct Members

Defining dictionaries as struct members is possible in Cairo but correctly interacting with them may not be entirely seamless. Let's try implementing a custom _user database_ that will allow us to add users and query them. We will need to define a struct to represent the new type and a trait to define its functionality:

```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_12_dict_struct_member/src/lib.cairo:struct}}
{{#include ../listings/ch11-advanced-features/no_listing_12_dict_struct_member/src/lib.cairo:trait}}
```

Our new type `UserDatabase<T>` represents a database of users. It is generic over the balances of the users, giving major flexibility to whoever uses our data type. Its two members are:

- `users_updates`, the number of users updates in the dictionary.
- `balances`, a mapping of each user to its balance.

The database core functionality is defined by `UserDatabaseTrait`. The following methods are defined:

- `new` for easily creating new `UserDatabase` types.
- `update_user` to update the balance of users in the database.
- `get_balance` to find user's balance in the database.

The only remaining step is to implement each of the methods in `UserDatabaseTrait`, but since we are working with [Generic types](./ch08-00-generic-types-and-traits.md) we also need to correctly establish the requirements of `T` so it can be a valid `Felt252Dict<T>` value type:
1. `T` should implement the `Copy<T>` since it's required for getting values from a `Felt252Dict<T>`.
2. All value types of a dictionary implement the `Felt252DictValue<T>`, our generic type should do as well.
3. To insert values, `Felt252DictTrait<T>` requires all value types to be droppable (implement the `Drop<T>` trait).
The implementation, with all restrictions in place, would be as follows:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_12_dict_struct_member/src/lib.cairo:impl}}
```
Our database implementation is almost complete, except for one thing: the compiler doesn't know how to make a `UserDatabase<T>` go out of scope, since it doesn't implement the `Drop<T>` trait, nor the `Destruct<T>` trait.
Since it has a `Felt252Dict<T>` as a member, it cannot be dropped, so we are forced to implement the `Destruct<T>` trait manually (refer to the [Ownership](ch04-01-what-is-ownership.md#the-drop-trait) chapter for more information).
Using `#[derive(Destruct)]` on top of the `UserDatabase<T>` definition won't work because of the use of [Generic types][generics] in the struct definition. We need to code the `Destruct<T>` trait implementation by ourselves:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_12_dict_struct_member/src/lib.cairo:destruct}}
```
Implementing `Destruct<T>` for `UserDatabase` was our last step to get a fully functional database. We can now try it out:
```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_12_dict_struct_member/src/lib.cairo:main}}
```
[generics]: ./ch08-00-generic-types-and-traits.md
Simulating a Dynamic Array with Dicts
First, let's think about how we want our mutable dynamic array to behave. What operations should it support?
It should:
- Allow us to append items at the end.
- Let us access any item by index.
- Allow setting the value of an item at a specific index.
- Return the current length.
We can define this interface in Cairo like:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_13_cust_struct_vect/src/
```

lib.cairo:trait}}
```
This provides a blueprint for the implementation of our dynamic array. We named it _Vec_ as it is similar to the `Vec<T>` data structure in Rust.
### Implementing a Dynamic Array in Cairo
To store our data, we'll use a `Felt252Dict<T>` which maps index numbers (felts) to values. We'll also store a separate `len` field to track the length.
Here is what our struct looks like. We wrap the type `T` inside `Nullable` pointer to allow using any type `T` in our data structure, as explained in the [Dictionaries][nullable] section:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_13_cust_struct_vect/src/lib.cairo:struct}}
```
Since we again have `Felt252Dict<T>` as a struct member, we need to implement the `Destruct<T>` trait to tell the compiler how to make `NullableVec<T>` go out of scope.
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_13_cust_struct_vect/src/lib.cairo:destruct}}
```
The key thing that makes this vector mutable is that we can insert values into the dictionary to set or update values in our data structure. For example, to update a value at a specific index, we do:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_13_cust_struct_vect/src/lib.cairo:set}}
```
This overwrites the previously existing value at that index in the dictionary. While arrays are immutable, dictionaries provide the flexibility we need for modifiable data structures like vectors.
The implementation of the rest of the interface is straightforward. The implementation of all the methods defined in our interface can be done as follow:
```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_13_cust_struct_vect/src/lib.cairo:implem}}
```
The full implementation of the `Vec` structure can be found in the community-maintained library [Alexandria](https://github.com/keep-starknet-strange/alexandria/tree/main/packages/data_structures/src).
[nullable]: ./ch03-02-dictionaries.md#dictionaries-of-types-not-supported-natively
Simulating a Stack with Dicts
We will now look at a second example and its implementation details: a Stack.
A Stack is a LIFO (Last-In, First-Out) collection. The insertion of a new element and removal of an existing element takes place at the same end, represented as the top of the stack.
```

Let us define what operations we need to create a stack:
- Push an item to the top of the stack.
- Pop an item from the top of the stack.
- Check whether there are still any elements in the stack.

From these specifications we can define the following interface :

```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_14_cust_struct_stack/src/lib.cairo:trait}}
```

Implementing a Mutable Stack in Cairo

To create a stack data structure in Cairo, we can again use a `Felt252Dict<T>` to store the values of the stack along with a `usize` field to keep track of the length of the stack to iterate over it.

The Stack struct is defined as:

```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_14_cust_struct_stack/src/lib.cairo:struct}}
```

Next, let's see how our main functions `push` and `pop` are implemented.

```cairo,noplayground
{{#include ../listings/ch11-advanced-features/no_listing_14_cust_struct_stack/src/lib.cairo:implem}}
```

The code uses the `insert` and `get` methods to access the values in the `Felt252Dict<T>`. To push an element to the top of the stack, the `push` function inserts the element in the dict at index `len` and increases the `len` field of the stack to keep track of the position of the stack top. To remove a value, the `pop` function decreases the value of `len` to update the position of the stack top and then retrieves the last value at position `len`.

The full implementation of the Stack, along with more data structures that you can use in your code, can be found in the community-maintained [Alexandria][alexandria data structures] library, in the "data_structures" crate.

[alexandria data structures]: https://github.com/keep-starknet-strange/alexandria/tree/main/packages/data_structures/src

{{#quiz ../quizzes/ch11-01-custom-structs.toml}}

Summary

Well done! Now you have knowledge of arrays, dictionaries and even custom data structures.

While Cairo's memory model is immutable and can make it difficult to implement mutable data structures, we can fortunately use the `Felt252Dict<T>` type to simulate mutable data structures. This allows us to implement a wide range of data structures that are useful for many applications, effectively hiding the complexity of the underlying memory model.