Generic Data Types

We use generics to create definitions for item declarations, such as structs and functions, which we can then use with many different concrete data types. In Cairo, we can use generics when defining functions, structs, enums, traits, implementations and methods. In this chapter, we are going to take a look at how to effectively use generic types with all of them.

Generics allow us to write reusable code that works with many types, thus avoiding code duplication, while enhancing code maintainability.

Generic Functions

Making a function generic means it can operate on different types, avoiding the need for multiple, type-specific implementations. This leads to significant code reduction and increases the flexibility of the code.

When defining a function that uses generics, we place the generics in the function signature, where we would usually specify the data types of the parameter and return value. For example, imagine we want to create a function which given two `Array` of items, will return the largest one. If we need to perform this operation for lists of different types, then we would have to redefine the function each time. Luckily we can implement the function once using generics and move on to other tasks.

```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_01_missing_tdrop/src/lib.cairo:generic}}
```

The `largest_list` function compares two lists of the same type and returns the one with more elements and drops the other. If you compile the previous code, you will notice that it will fail with an error saying that there are no traits defined for dropping an array of a generic type. This happens because the compiler has no way to guarantee that an `Array<T>` is droppable when executing the `main` function. In order to drop an array of `T`, the compiler must first know how to drop `T`. This can be fixed by specifying in the function signature of `largest_list` that `T` must implement the `Drop` trait. The correct function definition of `largest_list` is as follows:

```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_02_with_tdrop/src/lib.cairo}}
```

The new `largest_list` function includes in its definition the requirement that whatever generic type is placed there, it must be droppable. This is what we call _trait bounds_. The `main` function remains unchanged, the compiler is smart enough to deduce which concrete type is being used and if it implements the `Drop` trait.

Constraints for Generic Types

When defining generic types, it is useful to have information about them. Knowing which traits a generic type implements allows us to use it more effectively in a function's logic at the cost of constraining the generic types that can be used with the function. We saw an example of this previously by adding the `TDrop` implementation as part of the generic arguments of `largest_list`. While `TDrop` was added to satisfy the compiler's requirements, we can also add constraints to benefit our function logic. Imagine that we want, given a list of elements of some generic type `T`, to find the

smallest element among them. Initially, we know that for an element of type `T` to be comparable, it must implement the `PartialOrd` trait. The resulting function would be:
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_03_missing_tcopy/src/lib.cairo:missing-tcopy}}
```
The `smallest_element` function uses a generic type `T` that implements the `PartialOrd` trait, takes a snapshot of an `Array<T>` as a parameter and returns a copy of the smallest element. Because the parameter is of type `@Array<T>`, we no longer need to drop it at the end of the execution and so we are not required to implement the `Drop` trait for `T` as well. Why does it not compile then?
When indexing on `list`, the value results in a snap of the indexed element, and unless `PartialOrd` is implemented for `@T` we need to desnap the element using `*`. The `*` operation requires a copy from `@T` to `T`, which means that `T` needs to implement the `Copy` trait. After copying an element of type `@T` to `T`, there are now variables with type `T` that need to be dropped, requiring `T` to implement the `Drop` trait as well. We must then add both `Drop` and `Copy` traits implementation for the function to be correct. After updating the `smallest_element` function the resulting code would be:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_04_with_tcopy/src/lib.cairo}}
```
Anonymous Generic Implementation Parameter (`+` Operator)
Until now, we have always specified a name for each implementation of the required generic trait: `TPartialOrd` for `PartialOrd<T>`, `TDrop` for `Drop<T>`, and `TCopy` for `Copy<T>`.
However, most of the time, we don't use the implementation in the function body; we only use it as a constraint. In these cases, we can use the `+` operator to specify that the generic type must implement a trait without naming the implementation. This is referred to as an _anonymous generic implementation parameter_.
For example, `+PartialOrd<T>` is equivalent to `impl TPartialOrd: PartialOrd<T>`.
We can rewrite the `smallest_element` function signature as follows:
```cairo
{{#rustdoc_include ../listings/ch08-generic-types-and-traits/no_listing_05_with_anonymous_impl/src/lib.cairo:1}}
```
Structs
We can also define structs to use a generic type parameter for one or more fields using the `<>` syntax, similar to function definitions. First, we declare the name of the type parameter inside the angle brackets just after the name of the struct. Then we use the generic type in the struct definition where we would otherwise specify concrete data types. The next code example shows the definition `Wallet<T>` which has a `balance` field of type `T`.
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_06_derive_generics/src/lib.cairo}}
```

```
The above code derives the `Drop` trait for the `Wallet` type automatically. It is equivalent to writing the following code:
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_07_drop_explicit/src/lib.cairo}}
```
We avoid using the `derive` macro for `Drop` implementation of `Wallet` and instead define our own `WalletDrop` implementation. Notice that we must define, just like functions, an additional generic type for `WalletDrop` saying that `T` implements the `Drop` trait as well. We are basically saying that the struct `Wallet<T>` is droppable as long as `T` is also droppable.
Finally, if we want to add a field to `Wallet` representing its address and we want that field to be different than `T` but generic as well, we can simply add another generic type between the `<>`:
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_08_two_generics/src/lib.cairo}}
```
We add to the `Wallet` struct definition a new generic type `U` and then assign this type to the new field member `address`. Notice that the `derive` attribute for the `Drop` trait works for `U` as well.
Enums
As we did with structs, we can define enums to hold generic data types in their variants. For example the `Option<T>` enum provided by the Cairo core library:
```cairo,noplayground
{{#include ../listings/ch08-generic-types-and-traits/no_listing_09_option/src/lib.cairo}}
```
The `Option<T>` enum is generic over a type `T` and has two variants: `Some`, which holds one value of type `T` and `None` that doesn't hold any value. By using the `Option<T>` enum, it is possible for us to express the abstract concept of an optional value and because the value has a generic type `T` we can use this abstraction with any type.
Enums can use multiple generic types as well, like the definition of the `Result<T, E>` enum that the core library provides:
```cairo,noplayground
{{#include ../listings/ch08-generic-types-and-traits/no_listing_10_result/src/lib.cairo}}
```
The `Result<T, E>` enum has two generic types, `T` and `E`, and two variants: `Ok` which holds the value of type `T` and `Err` which holds the value of type `E`. This definition makes it convenient to use the `Result` enum anywhere we have an operation that might succeed (by returning a value of type `T`) or fail (by returning a value of type `E`).
Generic Methods
We can implement methods on structs and enums, and use the generic types in their definitions, too. Using our previous definition of `Wallet<T>` struct, we define a
```

`balance` method for it:
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_11_generic_methods/src/lib.cairo}}
```
We first define `WalletTrait<T>` trait using a generic type `T` which defines a method that returns the value of the field `balance` from `Wallet`. Then we give an implementation for the trait in `WalletImpl<T>`. Note that you need to include a generic type in both definitions of the trait and the implementation.
We can also specify constraints on generic types when defining methods on the type. We could, for example, implement methods only for `Wallet<u128>` instances rather than `Wallet<T>`. In the code example, we define an implementation for wallets which have a concrete type of `u128` for the `balance` field.
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_12_constrained_generics/src/lib.cairo}}
```
The new method `receive` increments the size of `balance` of any instance of a `Wallet<u128>`. Notice that we changed the `main` function making `w` a mutable variable in order for it to be able to update its balance. If we were to change the initialization of `w` by changing the type of `balance` the previous code wouldn't compile.
Cairo allows us to define generic methods inside generic traits as well. Using the past implementation from `Wallet<U, V>` we are going to define a trait that picks two wallets of different generic types and creates a new one with a generic type of each. First, let's rewrite the struct definition:
```cairo,noplayground
{{#include ../listings/ch08-generic-types-and-traits/no_listing_13_not_compiling/src/lib.cairo:struct}}
```
Next, we are going to naively define the mixup trait and implementation:
```cairo,noplayground
{{#include ../listings/ch08-generic-types-and-traits/no_listing_13_not_compiling/src/lib.cairo:trait_impl}}
```
We are creating a trait `WalletMixTrait<T1, U1>` with the `mixup<T2, U2>` method which given an instance of `Wallet<T1, U1>` and `Wallet<T2, U2>` creates a new `Wallet<T1, U2>`. As `mixup` signature specifies, both `self` and `other` are getting dropped at the end of the function, which is why this code does not compile. If you have been following from the start until now you would know that we must add a requirement for all the generic types specifying that they will implement the `Drop` trait for the compiler to know how to drop instances of `Wallet<T, U>`. The updated implementation is as follows:
```cairo
{{#include ../listings/ch08-generic-types-and-traits/no_listing_14_compiling/src/lib.cairo:trait_impl}}
```

```
We add the requirements for `T1` and `U1` to be droppable on `WalletMixImpl` declaration. Then we do the same for `T2` and `U2`, this time as part of `mixup` signature. We can now try the `mixup` function:
```cairo,noplayground
{{#include ../listings/ch08-generic-types-and-traits/no_listing_14_compiling/src/lib.cairo:main}}
```
We first create two instances: one of `Wallet<bool, u128>` and the other of `Wallet<felt252, u8>`. Then, we call `mixup` and create a new `Wallet<bool, u8>` instance.
```