Deref Coercion
Deref coercion simplifies the way we interact with nested or wrapped data structures by allowing an instance of one type to behave like an instance of another type. This mechanism is enabled by implementing the `Deref` trait, which allows implicit conversion (or coercion) to a different type, providing direct access to the underlying data.
> Note: For now, deref coercion allows you to access a member of a type `T` as if it was a type `K`, but will not allow you to call functions whose `self` argument is of the original type when holding an instance of the coerced type.
Deref coercion is implemented via the `Deref` and `DerefMut` traits. When a type `T` implements `Deref` or `DerefMut` to type `K`, instances of `T` can access the members of `K` directly.
The `Deref` trait in Cairo is defined as follows:
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/listing_09_deref_coercion/src/lib.cairo:Deref}}
```
The `Target` type specifies the result of dereferencing, and the `deref` method defines how to transform `T` into `K`.
Using Deref Coercion
To better understand how deref coercion works, let's look at a practical example. We'll create a simple generic wrapper type around a type `T` called `Wrapper<T>`, and use it to wrap a `UserProfile` struct.
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_09_deref_example/src/lib.cairo:Wrapper}}
```
The `Wrapper` struct wraps a single value generic of type `T`. To simplify access to the wrapped value, we implement the `Deref` trait for `Wrapper<T>`.
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_09_deref_example/src/lib.cairo:deref}}
```
This implementation is quite simple. The `deref` method returns the wrapped value, allowing instances of `Wrapper<T>` to access the members of `T` directly.
In practice, this mechanism is totally transparent. The following example demonstrates how, holding
an instance of `Wrapper<UserProfile>`, we can print the `username` and `age` fields of the underlying
`UserProfile` instance.
```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/no_listing_09_deref_example/src/lib.cairo:main}}
```
Restricting Deref Coercion to Mutable Variables
While `Deref` works for both mutable and immutable variables, `DerefMut` will only be

applicable to mutable variables. Contrary to what the name might suggest, `DerefMut` does not provide mutable access to the underlying data.
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/
no_listing_09_deref_mut_example/src/lib.cairo:derefMut}}
```
If you try to use `DerefMut` with an immutable variable, the compiler would throw an error. Here's an example:
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/
no_listing_09_deref_mut_example/src/lib.cairo:error}}
```
Compiling this code will result in the following error:
```plaintext
{{#rustdoc_include ../listings/ch11-advanced-features/
no_listing_09_deref_mut_example/output.txt}}
```
For the above code to work, we need to define `wrapped_profile` as a mutable variable.
```cairo, noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/
no_listing_09_deref_mut_example/src/lib.cairo:example}}
```
Summary
By using the `Deref` and `DerefMut` traits, we can transparently convert one type into another, simplifying the access to nested or wrapped data structures. This feature is particularly useful when working with generic types or building abstractions that require seamless access to the underlying data and can help reduce boilerplate code. However, this functionality is quite limited, as you cannot call functions whose `self` argument is of the original type when holding an instance of the coerced type.