# Ownership Using a Linear Type System
Cairo uses a linear type system. In such a type system, any value (a basic type, a struct, an enum) must be used and must only be used once. 'Used' here means that the value is either _destroyed_ or _moved_.
_Destruction_ can happen in several ways:
- a variable goes out of scope.
- a struct is destructured.
- explicit destruction using `destruct()`.
_Moving_ a value simply means passing that value to another function.
This results in somewhat similar constraints to the Rust ownership model, but there are some differences.
In particular, the Rust ownership model exists (in part) to avoid data races and concurrent mutable access to a memory value. This is obviously impossible in Cairo since the memory is immutable.
Instead, Cairo leverages its linear type system for two main purposes:
- Ensuring that all code is provable and thus verifiable.
- Abstracting away the immutable memory of the Cairo VM.
### Ownership
In Cairo, ownership applies to _variables_ and not to _values_. A value can safely be referred to by many different variables (even if they are mutable variables), as the value itself is always immutable.
Variables however can be mutable, so the compiler must ensure that constant variables aren't accidentally modified by the programmer.
This makes it possible to talk about ownership of a variable: the owner is the code that can read (and write if mutable) the variable.
This means that variables (not values) follow similar rules to Rust values:
- Each variable in Cairo has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the variable is destroyed.
Now that we’re past basic Cairo syntax, we won’t include all the `fn main() {` code in examples, so if you’re following along, make sure to put the following examples inside a main function manually. As a result, our examples will be a bit more concise, letting us focus on the actual details rather than boilerplate code.
## Variable Scope
As a first example of the linear type system, we’ll look at the _scope_ of some variables. A
scope is the range within a program for which an item is valid. Take the
following variable:
```cairo,noplayground
let s = 'hello';
```
The variable `s` refers to a short string. The variable is valid from the point at
which it’s declared until the end of the current _scope_. Listing {{#ref variable-scope}} shows a
program with comments annotating where the variable `s` would be valid.
```cairo
```

{{#rustdoc_include ../listings/ch04-understanding-ownership/
listing_01_variable_and_scope/src/lib.cairo:here}}
```

{{#label variable-scope}}
<span class="caption">Listing {{#ref variable-scope}}: A variable and the scope in which it is valid</span>
In other words, there are two important points in time here:
- When `s` comes _into_ scope, it is valid.
- It remains valid until it goes _out of_ scope.
At this point, the relationship between scopes and when variables are valid is similar to that in other programming languages. Now we'll build on top of this understanding by using the `Array` type we introduced in the previous ["Arrays"][array] section.
[array]: ./ch03-01-arrays.md
### Moving values
As said earlier, _moving_ a value simply means passing that value to another function. When that happens, the variable referring to that value in the original scope is destroyed and can no longer be used, and a new variable is created to hold the same value.
Arrays are an example of a complex type that is moved when passing it to another function.
Here is a short reminder of what an array looks like:
```cairo
{{#rustdoc_include ../listings/ch04-understanding-ownership/no_listing_01_array/src/
lib.cairo:2:4}}
```

How does the type system ensure that the Cairo program never tries to write to the same memory cell twice?
Consider the following code, where we try to remove the front of the array twice:
```cairo,does_not_compile
{{#include ../listings/ch04-understanding-ownership/no_listing_02_pass_array_by_value/
src/lib.cairo}}
```

In this case, we try to pass the same value (the array in the `arr` variable) to both function calls. This means our code tries to remove the first element twice, which would try to write to the same memory cell twice - which is forbidden by the Cairo VM, leading to a runtime error.
Thankfully, this code does not actually compile. Once we have passed the array to the `foo` function, the variable `arr` is no longer usable. We get this compile-time error, telling us that we would need Array to implement the Copy Trait:
```shell
{{#include ../listings/ch04-understanding-ownership/no_listing_02_pass_array_by_value/
output.txt}}
```

## The `Copy` Trait
The `Copy` trait allows simple types to be duplicated by copying felts, without allocating new memory segments. This contrasts with Cairo's default "move" semantics, which
```

transfer ownership of values to ensure memory safety and prevent issues like multiple writes to the same memory cell. `Copy` is implemented for types where duplication is safe and efficient, bypassing the need for move semantics. Types like `Array` and `Felt252Dict` cannot implement `Copy`, as manipulating them in different scopes is forbidden by the type system.
All basic types previously described in ["Data Types"][data types] implement by default the `Copy` trait.
While Arrays and Dictionaries can't be copied, custom types that don't contain either of them can be.
You can implement the `Copy` trait on your type by adding the `#[derive(Copy)]` annotation to your type definition. However, Cairo won't allow a type to be annotated with Copy if the type itself or any of its components doesn't implement the Copy trait.
```cairo,ignore_format
{{#include ../listings/ch04-understanding-ownership/no_listing_03_copy_trait/src/lib.cairo}}
```
In this example, we can pass `p1` twice to the foo function because the `Point` type implements the `Copy` trait. This means that when we pass `p1` to `foo`, we are actually passing a copy of `p1`, so `p1` remains valid. In ownership terms, this means that the ownership of `p1` remains with the `main` function.
If you remove the `Copy` trait derivation from the `Point` type, you will get a compile-time error when trying to compile the code.
_Don't worry about the `Struct` keyword. We will introduce this in [Chapter 5][structs]._
[data types]: ./ch02-02-data-types.md
[structs]: ./ch05-00-using-structs-to-structure-related-data.md
## Destroying Values - Example with FeltDict
The other way linear types can be _used_ is by being destroyed. Destruction must ensure that the 'resource' is now correctly released. In Rust, for example, this could be closing the access to a file, or locking a mutex.
In Cairo, one type that has such behaviour is `Felt252Dict`. For provability, dicts must be 'squashed' when they are destructed.
This would be very easy to forget, so it is enforced by the type system and the compiler.
### No-op Destruction: the `Drop` Trait
You may have noticed that the `Point` type in the previous example also implements the `Drop` trait.
For example, the following code will not compile, because the struct `A` is not moved or destroyed before it goes out of scope:
```cairo,does_not_compile
{{#include ../listings/ch04-understanding-ownership/no_listing_04_no_drop_derive_fails/src/lib.cairo}}
```
However, types that implement the `Drop` trait are automatically destroyed when going out of scope. This destruction does nothing, it is a no-op - simply a hint to the compiler that this type can safely be destroyed once it's no longer useful. We call this "dropping" a value.
At the moment, the `Drop` implementation can be derived for all types, allowing them to

be dropped when going out of scope, except for dictionaries (`Felt252Dict`) and types containing dictionaries.
For example, the following code compiles:
```cairo
{{#include ../listings/ch04-understanding-ownership/
no_listing_05_drop_derive_compiles/src/lib.cairo}}
```
### Destruction with a Side-effect: the `Destruct` Trait
When a value is destroyed, the compiler first tries to call the `drop` method on that type. If it doesn't exist, then the compiler tries to call `destruct` instead. This method is provided by the `Destruct` trait.
As said earlier, dictionaries in Cairo are types that must be "squashed" when destructed, so that the sequence of access can be proven. This is easy for developers to forget, so instead dictionaries implement the `Destruct` trait to ensure that all dictionaries are _squashed_ when going out of scope.
As such, the following example will not compile:
```cairo,does_not_compile
{{#include ../listings/ch04-understanding-ownership/
no_listing_06_no_destruct_compile_fails/src/lib.cairo}}
```
If you try to run this code, you will get a compile-time error:
```shell
{{#include ../listings/ch04-understanding-ownership/
no_listing_06_no_destruct_compile_fails/output.txt}}
```
When `A` goes out of scope, it can't be dropped as it implements neither the `Drop` (as it contains a dictionary and can't `derive(Drop)`) nor the `Destruct` trait. To fix this, we can derive the `Destruct` trait implementation for the `A` type:
```cairo
{{#include ../listings/ch04-understanding-ownership/no_listing_07_destruct_compiles/
src/lib.cairo}}
```
Now, when `A` goes out of scope, its dictionary will be automatically `squashed`, and the program will compile.
## Copy Array Data with `clone`
If we _do_ want to deeply copy the data of an `Array`, we can use a common method called `clone`. We'll discuss method syntax in a dedicated section in [Chapter 5][method syntax], but because methods are a common feature in many programming languages, you've probably seen them before.
Here's an example of the `clone` method in action.
```cairo
{{#include ../listings/ch04-understanding-ownership/no_listing_08_array_clone/src/
lib.cairo}}
```
When you see a call to `clone`, you know that some arbitrary code is being executed and that code may be expensive. It's a visual indicator that something different is going

on.
In this case, the _value_ `arr1` refers to is being copied, resulting in new memory cells being used, and a new _variable_ `arr2` is created, referring to the new copied value.
[method syntax]: ./ch05-03-method-syntax.md
## Return Values and Scope
Returning values is equivalent to _moving_ them. Listing {{#ref move-return-values}} shows an example of a
function that returns some value, with similar annotations as those in Listing {{#ref variable-scope}}.
Filename: src/lib.cairo
```cairo
{{#include ../listings/ch04-understanding-ownership/listing_02_moving_return_values/src/lib.cairo}}
```
{{#label move-return-values}}
Listing {{#ref move-return-values}}: Moving return values
While this works, moving into and out of every function is a bit tedious. What if we want to let a function use a value but not move the value? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.
Cairo does let us return multiple values using a tuple, as shown in Listing {{#ref return-multiple-values}}.
Filename: src/lib.cairo
```cairo
{{#include ../listings/ch04-understanding-ownership/listing_03_returning_many_values/src/lib.cairo}}
```
{{#label return-multiple-values}}
Listing {{#ref return-multiple-values}}: Returning many values
But this is too much ceremony and a lot of work for a concept that should be common. Luckily for us, Cairo has two features for passing a value without destroying or moving it, called _references_ and _snapshots_.