Variables and Mutability

Cairo uses an immutable memory model, meaning that once a memory cell is written to, it can't be overwritten but only read from. To reflect this immutable memory model, variables in Cairo are immutable by default.

However, the language abstracts this model and gives you the option to make your variables mutable. Let’s explore how and why Cairo enforces immutability, and how you can make your variables mutable.

When a variable is immutable, once a value is bound to a name, you can’t change that value. To illustrate this, generate a new project called _variables_ in your _cairo_projects_ directory by using `scarb new variables`.

Then, in your new _variables_ directory, open _src/lib.cairo_ and replace its code with the following code, which won’t compile just yet:

Filename: src/lib.cairo

```cairo,does_not_compile
{{#include ../listings/ch02-common-programming-concepts/
no_listing_01_variables_are_immutable/src/lib.cairo}}
```

Save and run the program using `scarb cairo-run`. You should receive an error message regarding an immutability error, as shown in this output:

```shell
{{#include ../listings/ch02-common-programming-concepts/
no_listing_01_variables_are_immutable/output.txt}}
```

This example shows how the compiler helps you find errors in your programs. Compiler errors can be frustrating, but they only mean your program isn’t safely doing what you want it to do yet; they do _not_ mean that you’re not a good programmer! Experienced Caironautes still get compiler errors.

You received the error message `Cannot assign to an immutable variable.` because you tried to assign a second value to the immutable `x` variable.

It’s important that we get compile-time errors when we attempt to change a value that’s designated as immutable because this specific situation can lead to bugs. If one part of our code operates on the assumption that a value will never change and another part of our code changes that value, it’s possible that the first part of the code won’t do what it was designed to do. The cause of this kind of bug can be difficult to track down after the fact, especially when the second piece of code changes the value only _sometimes_.

Cairo, unlike most other languages, has immutable memory. This makes a whole class of bugs impossible, because values will never change unexpectedly. This makes code easier to reason about.

But mutability can be very useful, and can make code more convenient to write. Although variables are immutable by default, you can make them mutable by adding `mut` in front of the variable name. Adding `mut` also conveys intent to future readers of the code by indicating that other parts of the code will be changing the value associated to this variable.

<!-- TODO: add an illustration of this -->

However, you might be wondering at this point what exactly happens when a variable

is declared as `mut`, as we previously mentioned that Cairo's memory is immutable. The answer is that the _value_ is immutable, but the _variable_ isn't. The value associated to the variable can be changed. Assigning to a mutable variable in Cairo is essentially equivalent to redeclaring it to refer to another value in another memory cell,
but the compiler handles that for you, and the keyword `mut` makes it explicit.
Upon examining the low-level Cairo Assembly code, it becomes clear that variable mutation is implemented as syntactic sugar, which translates mutation operations
into a series of steps equivalent to variable shadowing. The only difference is that at the Cairo
level, the variable is not redeclared so its type cannot change.
For example, let's change _src/lib.cairo_ to the following:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_02_adding_mut/src/lib.cairo}}
```
When we run the program now, we get this:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_02_adding_mut/output.txt}}
```
We're allowed to change the value bound to `x` from `5` to `6` when `mut` is used. Ultimately, deciding whether to use mutability or not is up to you and depends on what you think is clearest in that particular situation.
Constants
Like immutable variables, _constants_ are values that are bound to a name and are not allowed to change, but there are a few differences between constants and variables.
First, you aren't allowed to use `mut` with constants. Constants aren't just immutable by default—they're always immutable. You declare constants using the `const` keyword instead of the `let` keyword, and the type of the value _must_ be annotated. We'll cover types and type annotations in the next section, ["Data Types"][data-types], so don't worry about the details
right now. Just know that you must always annotate the type.
Constant variables can be declared with any usual data type, including structs, enums and fixed-size arrays.
Constants can only be declared in the global scope, which makes
them useful for values that many parts of code need to know about.
The last difference is that constants may natively be set only to a constant expression, not the result of a value that could only be computed at runtime.
Here's an example of constants declaration:
```cairo,noplayground
{{#include ../listings/ch02-common-programming-concepts/no_listing_00_consts/src/lib.cairo:const_expressions}}
```

Nonetheless, it is possible to use the `consteval_int!` macro to create a `const` variable that is the result of some computation:
```cairo, noplayground
{{#include ../listings/ch02-common-programming-concepts/no_listing_00_consts/src/lib.cairo:consteval_const}}
```
We will dive into more detail about macros in the [dedicated section](./ch11-05-macros.md).
Cairo's naming convention for constants is to use all uppercase with underscores between words.
Constants are valid for the entire time a program runs, within the scope in which they were declared. This property makes constants useful for values in your application domain that multiple parts of the program might need to know about, such as the maximum number of points any player of a game is allowed to earn, or the speed of light.
Naming hardcoded values used throughout your program as constants is useful in conveying the meaning of that value to future maintainers of the code. It also helps to have only one place in your code you would need to change if the hardcoded value needed to be updated in the future.
[data-types]: ./ch02-02-data-types.md
Shadowing
Variable shadowing refers to the declaration of a
new variable with the same name as a previous variable. Caironautes say that the first variable is _shadowed_ by the second, which means that the second variable is what the compiler will see when you use the name of the variable.
In effect, the second variable overshadows the first, taking any uses of the variable name to itself until either it itself is shadowed or the scope ends.
We can shadow a variable by using the same variable's name and repeating the use of the `let` keyword as follows:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_03_shadowing/src/lib.cairo}}
```
This program first binds `x` to a value of `5`. Then it creates a new variable `x` by repeating `let x =`, taking the original value and adding `1` so the value of `x` is then `6`. Then, within an inner scope created with the curly brackets, the third `let` statement also shadows `x` and creates a new variable, multiplying the previous value by `2` to give `x` a value of `12`.
When that scope is over, the inner shadowing ends and `x` returns to being `6`.
When we run this program, it will output the following:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_03_shadowing/output.txt}}
```
Shadowing is different from marking a variable as `mut` because we'll get a compile-time error if we accidentally try to reassign to this variable without

using the `let` keyword. By using `let`, we can perform a few transformations
on a value but have the variable be immutable after those transformations have
been completed.
Another distinction between `mut` and shadowing is that when we use the `let` keyword
again,
we are effectively creating a new variable, which allows us to change the type of the
value while reusing the same name. As mentioned before, variable shadowing and
mutable variables
are equivalent at the lower level.
The only difference is that by shadowing a variable, the compiler will not complain
if you change its type. For example, say our program performs a type conversion
between the
`u64` and `felt252` types.
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_04_shadowing_different_type/src/lib.cairo}}
```
The first `x` variable has a `u64` type while the second `x` variable has a `felt252` type.
Shadowing thus spares us from having to come up with different names, such as
`x_u64`
and `x_felt252`; instead, we can reuse the simpler `x` name. However, if we try to use
`mut` for this, as shown here, we'll get a compile-time error:
```cairo,does_not_compile
{{#include ../listings/ch02-common-programming-concepts/
no_listing_05_mut_cant_change_type/src/lib.cairo}}
```
The error says we were expecting a `u64` (the original type) but we got a different type:
```shell
{{#include ../listings/ch02-common-programming-concepts/
no_listing_05_mut_cant_change_type/output.txt}}
```
{{#quiz ../quizzes/ch02-01-variables-and-mutability.toml}}
Now that we've explored how variables work, let's look at more data types they can
have.