The Match Control Flow Construct
Cairo has an extremely powerful control flow construct called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards, and many other things. The power of `match` comes from the expressiveness of the patterns and the fact that the compiler confirms that all possible cases are handled.
Think of a `match` expression as being like a coin-sorting machine: coins slide down a track with variously sized holes along it, and each coin falls through the first hole it encounters that it fits into. In the same way, values go through each pattern in a match, and at the first pattern the value “fits”, the value falls into the associated code block to be used during execution.
Speaking of coins, let’s use them as an example using `match`! We can write a function that takes an unknown US coin and, in a similar way as the counting machine, determines which coin it is and returns its value in cents, as shown in Listing {{#ref match-enum}}.
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_05_match_example/src/lib.cairo:all}}
```
{{#label match-enum}}
<span class="caption">Listing {{#ref match-enum}}: An enum and a `match` expression that has the variants of the enum as its patterns</span>
Let’s break down the `match` expression in the `value_in_cents` function. First, we list the `match` keyword followed by an expression, which in this case is the value `coin`. This seems very similar to a conditional expression used with the `if` statement, but there’s a big difference: with `if`, the condition needs to evaluate to a boolean value, but here it can be any type. The type of `coin` in this example is the `Coin` enum that we defined on the first line.
Next are the `match` arms. An arm has two parts: a pattern and some code. The first arm here has a pattern that is the value `Coin::Penny` and then the `=>` operator that separates the pattern and the code to run. The code in this case is just the value `1`. Each arm is separated from the next with a comma.
When the `match` expression executes, it compares the resultant value against the pattern of each arm, in the order they are given. If a pattern matches the value, the code associated with that pattern is executed. If that pattern doesn’t match the value, execution continues to the next arm, much as in a coin-sorting machine. We can have as many arms as we need: in the above example, our `match` has four arms.
The code associated with each arm is an expression, and the resultant value of the expression in the matching arm is the value that gets returned for the entire match expression.
We don’t typically use curly brackets if the `match` arm code is short, as it is in our example where each arm just returns a value. If you want to run multiple lines of code in a `match` arm, you must use curly brackets, with a comma following the arm. For example, the following code prints “Lucky penny!” every time the method is called with a `Coin::Penny`, but still returns the last value of the block, `1`:
```cairo,noplayground
```

{{#include ../listings/ch06-enums-and-pattern-matching/
no_listing_06_match_arms_block/src/lib.cairo:here}}
~~~
## Patterns That Bind to Values
Another useful feature of `match` arms is that they can bind to the parts of the values that match the pattern. This is how we can extract values out of enum variants.
As an example, let’s change one of our enum variants to hold data inside it. From 1999 through 2008, the United States minted quarters with different designs for each of the 50 states on one side. No other coins got state designs, so only quarters have this extra value. We can add this information to our `enum` by changing the `Quarter` variant to include a `UsState` value stored inside it, which we’ve done in Listing {{#ref match-pattern-bind}}.
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/
no_listing_07_match_pattern_bind/src/lib.cairo:enum_def}}
~~~
{{#label match-pattern-bind}}
<span class="caption">Listing {{#ref match-pattern-bind}}: A `Coin` enum in which the `Quarter` variant also holds a `UsState` value</span>
Let’s imagine that a friend is trying to collect all 50 state quarters. While we sort our loose change by coin type, we’ll also call out the name of the state associated with each quarter so that if it’s one our friend doesn’t have, they can add it to their collection.
In the `match` expression for this code, we add a variable called `state` to the pattern that matches values of the variant `Coin::Quarter`. When a `Coin::Quarter` matches, the `state` variable will bind to the value of that quarter’s state. Then we can use `state` in the code for that arm, like so:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/
no_listing_07_match_pattern_bind/src/lib.cairo:function}}
```
Because `state` is an `UsState` enum which implements the `Debug` trait, we can print `state` value with `println!` macro.
> Note: `{:?}` is a special formatting syntax that allows to print a debug form of the parameter passed to the `println!` macro. You can find more information about it in [Appendix C][debug trait].

If we were to call `value_in_cents(Coin::Quarter(UsState::Alaska))`, `coin` would be `Coin::Quarter(UsState::Alaska)`. When we compare that value with each of the match arms, none of them match until we reach `Coin::Quarter(state)`. At that point, the binding for `state` will be the value `UsState::Alaska`. We can then use that binding in `println!` macro, thus getting the inner state value out of the `Coin` enum variant for `Quarter`.
[debug trait]: ./appendix-03-derivable-traits.md#debug-for-printing-and-debugging
Matching with `Option<T>`
In the previous section, we wanted to get the inner `T` value out of the `Some` case when using `Option<T>`; we can also handle `Option<T>` using `match`, as we did with the `Coin` enum! Instead of comparing coins, we’ll compare the variants of

`Option<T>`, but the way the `match` expression works remains the same.
Let’s say we want to write a function that takes an `Option<u8>` and, if there’s a value inside, adds `1` to that value. If there is no value inside, the function should return the `None` value and not attempt to perform any operations.
This function is very easy to write, thanks to `match`, and will look like Listing {{#ref match-option}}.
```cairo
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_08_match_option/src/lib.cairo:all}}
```
{{#label match-option}}
<span class="caption">Listing {{#ref match-option}}: A function that uses a `match` expression on an `Option<u8>`</span>
Let’s examine the first execution of `plus_one` in more detail. When we call `plus_one(five)`, the variable `x` in the body of `plus_one` will have the value `Some(5)`. We then compare that against each `match` arm:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_08_match_option/src/lib.cairo:option_some}}
```
Does `Option::Some(5)` value match the pattern `Option::Some(val)`? It does! We have the same variant. The `val` binds to the value contained in `Option::Some`, so `val` takes the value `5`. The code in the `match` arm is then executed, so we add `1` to the value of `val` and create a new `Option::Some` value with our total `6` inside. Because the first arm matched, no other arms are compared.
Now let’s consider the second call of `plus_one` in our main function, where `x` is `Option::None`. We enter the `match` and compare to the first arm:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_08_match_option/src/lib.cairo:option_some}}
```
The `Option::Some(val)` value doesn’t match the pattern `Option::None`, so we continue to the next arm:
```cairo
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_08_match_option/src/lib.cairo:option_none}}
```
It matches! There’s no value to add to, so the matching construct ends and returns the `Option::None` value on the right side of `=>`.
Combining `match` and enums is useful in many situations. You’ll see this pattern a lot in Cairo code: `match` against an enum, bind a variable to the data inside, and then execute code based on it. It’s a bit tricky at first, but once you get used to it, you’ll wish you had it in all languages. It’s consistently a user favorite.
Matches Are Exhaustive
There’s one other aspect of `match` we need to discuss: the arms’ patterns must cover all possibilities. Consider this version of our `plus_one` function, which has a bug and

won’t compile:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/
no_listing_09_missing_match_arm/src/lib.cairo:here}}
```
We didn’t handle the `None` case, so this code will cause a bug.
Luckily, it’s a bug Cairo knows how to catch. If we try to compile this code, we’ll get this error:
```shell
{{#include ../listings/ch06-enums-and-pattern-matching/
no_listing_09_missing_match_arm/output.txt}}
```
Cairo knows that we didn’t cover every possible case, and even knows which pattern we forgot! Matches in Cairo are exhaustive: we must exhaust every last possibility in order for the code to be valid. Especially in the case of `Option<T>`, when Cairo prevents us from forgetting to explicitly handle the `None` case, it protects us from assuming that we have a value when we might have null, thus making the [billion-dollar mistake][null pointer] discussed earlier impossible.
[null pointer]: https://en.wikipedia.org/wiki/Null_pointer#History
Catch-all with the `_` Placeholder
Using enums, we can also take special actions for a few particular values, but for all other values take one default action.
`_` is a special pattern that matches any value and does not bind to that value.
You can use it by simply adding a new arm with `_` as the pattern for the last arm of the `match` expression.
Imagine we have a vending machine that only accepts Dime coins.
We want to have a function that processes inserted coins and returns `true` only if the coin is accepted.
Here's a `vending_machine_accept` function that implements this logic:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_10_match_catch_all/
src/lib.cairo:here}}
```
This example also meets the exhaustiveness requirement because we’re explicitly ignoring all other values in the last arm; we haven’t forgotten anything.
> There's no catch-all pattern in Cairo that allows you to use the value of the pattern.
<!--
  TODO move the following in a separate chapter when there's more pattern matching features in upcoming Cairo versions. cf rust book chapter 18
-->
Multiple Patterns with the `|` Operator
In `match` expressions, you can match multiple patterns using the `|` syntax, which is the pattern _or_ operator.
For example, in the following code we modified the `vending_machine_accept` function to accept both `Dime` and `Quarter` coins in a single arm:
```cairo,noplayground
```

{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_11_match_or/src/lib.cairo:here}}
~~~
## Matching Tuples
It is possible to match tuples.
Let's introduce a new `DayType` enum:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_12_match_tuple/src/lib.cairo:enum_def}}
~~~
Now, let's suppose that our vending machine accepts any coin on weekdays, but only accepts quarters and dimes on weekends and holidays.
We can modify the `vending_machine_accept` function to accept a tuple of a `Coin` and a `Weekday` and return `true` only if the given coin is accepted on the specified day:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_12_match_tuple/src/lib.cairo:here}}
```
Writing `(_, _)` for the last arm of a tuple matching pattern might feel superfluous.
Hence, we can use the `_ =>` syntax if we want, for example, that our vending machine only accepts quarters on weekdays:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_12_match_tuple/src/lib.cairo:week}}
```
Matching `felt252` and Integer Variables
You can also match `felt252` and integer variables. This is useful when you want to match against a range of values.
However, there are some restrictions:
- Only integers that fit into a single `felt252` are supported (i.e. `u256` is not supported).
- The first arm must be 0.
- Each arm must cover a sequential segment, contiguously with other arms.
Imagine we’re implementing a game where you roll a six-sided die to get a number between 0 and 5.
If you have 0, 1 or 2 you win. If you have 3, you can roll again. For all other values you lose.
Here's a match that implements that logic:
```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_13_match_integers/src/lib.cairo:here}}
```
{{#quiz ../quizzes/ch06-02-match.toml}}
> These restrictions are planned to be relaxed in future versions of Cairo.