Data Types
Every value in Cairo is of a certain _data type_, which tells Cairo what kind of
data is being specified so it knows how to work with that data. This section covers two subsets of data types: scalars and compounds.
Keep in mind that Cairo is a _statically typed_ language, which means that it
must know the types of all variables at compile time. The compiler can usually infer the desired type based on the value and its usage. In cases when many types are possible, we can use a conversion method where we specify the desired output type.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_06_data_types/src/lib.cairo}}
```
You’ll see different type annotations for other data types.
Scalar Types
A _scalar_ type represents a single value. Cairo has three primary scalar types:
felts, integers, and booleans. You may recognize
these from other programming languages. Let’s jump into how they work in Cairo.
Felt Type
In Cairo, if you don't specify the type of a variable or argument, its type defaults to a field element, represented by the keyword `felt252`. In the context of Cairo, when we say “a field element” we mean an integer in the range \\(0 \leq x < P \\),
where \\(P \\) is a very large prime number currently equal to \\({2^{251}} + 17 \cdot {2^{192}} + 1 \\). When adding, subtracting, or multiplying, if the result falls outside the specified range of the prime number, an overflow (or underflow) occurs, and an appropriate multiple of \\(P \\) is added or subtracted to bring the result back within the range (i.e., the result is computed \\(\mod P \\)).
The most important difference between integers and field elements is division: Division of field elements (and therefore division in Cairo) is unlike regular CPUs division, where integer division \\(\frac{x}{y} \\) is defined as \\(\left\lfloor \frac{x}{y} \right\rfloor \\) where the integer part of the quotient is returned (so you get \\(\frac{7}{3} = 2 \\)) and it may or may not satisfy the equation \\(\frac{x}{y} \cdot y == x \\),
depending on the divisibility of `x` by `y`.
In Cairo, the result of \\(\frac{x}{y} \\) is defined to always satisfy the equation \\(\frac{x}{y} \cdot y == x \\). If y divides x as integers, you will get the expected result in Cairo (for example \\(\frac{6}{2} \\) will indeed result in `3`).
But when y does not divide x, you may get a surprising result: for example, since \\(2 \cdot \frac{P + 1}{2} = P + 1 \equiv 1 \mod P \\), the value of \\(\frac{1}{2} \\) in Cairo is \\(\frac{P + 1}{2} \\) (and not 0 or 0.5), as it satisfies the above equation.
Integer Types
The felt252 type is a fundamental type that serves as the basis for creating all types in the core library.
However, it is highly recommended for programmers to use the integer types instead of the `felt252` type whenever possible, as the `integer` types come with added security features that provide extra protection against potential vulnerabilities in the code, such as overflow and underflow checks. By using these integer types, programmers can ensure that their programs are more secure and less susceptible to attacks or other

security threats.
An `integer` is a number without a fractional component. This type declaration indicates the number of bits the programmer can use to store the integer.
Table 3-1 shows the built-in integer types in Cairo. We can use any of these variants to declare the type of an integer value.

Length	Unsigned
8-bit	`u8`
16-bit	`u16`
32-bit	`u32`
64-bit	`u64`
128-bit	`u128`
256-bit	`u256`
32-bit	`usize`

<br>
<div align="center"><span class="caption">Table 3-1: Integer Types in Cairo.</span></div>
Each variant has an explicit size. Note that for now, the `usize` type is just an alias for `u32`; however, it might be useful when in the future Cairo can be compiled to MLIR.
As variables are unsigned, they can't contain a negative number. This code will cause the program to panic:

```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_07_integer_types/src/lib.cairo}}
```

All integer types previously mentioned fit into a `felt252`, except for `u256` which needs 4 more bits to be stored. Under the hood, `u256` is basically a struct with 2 fields: `u256 {low: u128, high: u128}`.
Cairo also provides support for signed integers, starting with the prefix `i`. These integers can represent both positive and negative values, with sizes ranging from `i8` to `i128`.
Each signed variant can store numbers from $-({2^{n - 1}})$ to ${2^{n - 1}} - 1$ inclusive, where `n` is the number of bits that variant uses. So an i8 can store numbers from $-({2^7})$ to ${2^7} - 1$, which equals `-128` to `127`.
You can write integer literals in any of the forms shown in Table 3-2. Note
that number literals that can be multiple numeric types allow a type suffix,
such as `57_u8`, to designate the type.
It is also possible to use a visual separator `_` for number literals, in order to improve code readability.

Numeric literals	Example
Decimal	`98222`
Hex	`0xff`
Octal	`0o04321`
Binary	`0b01`

<br>

<div align="center"><span class="caption">Table 3-2: Integer Literals in Cairo.</span></div>
So how do you know which type of integer to use? Try to estimate the max value your int can have and choose the good size.
The primary situation in which you’d use `usize` is when indexing some sort of collection.
Numeric Operations
Cairo supports the basic mathematical operations you’d expect for all the integer types: addition, subtraction, multiplication, division, and remainder. Integer division truncates toward zero to the nearest integer. The following code shows how you’d use each numeric operation in a `let` statement:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_08_numeric_operations/src/lib.cairo}}
```
Each expression in these statements uses a mathematical operator and evaluates to a single value, which is then bound to a variable.
[Appendix B][operators] contains a list of all operators that Cairo provides.
[operators]: ./appendix-02-operators-and-symbols.md#operators
The Boolean Type
As in most other programming languages, a Boolean type in Cairo has two possible values: `true` and `false`. Booleans are one `felt252` in size. The Boolean type in Cairo is specified using `bool`. For example:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_09_boolean_type/src/lib.cairo}}
```
When declaring a `bool` variable, it is mandatory to use either `true` or `false` literals as value. Hence, it is not allowed to use integer literals (i.e. `0` instead of false) for `bool` declarations.
The main way to use Boolean values is through conditionals, such as an `if` expression. We’ll cover how `if` expressions work in Cairo in the ["Control Flow"][control-flow] section.
[control-flow]: ./ch02-05-control-flow.md
String Types
Cairo doesn't have a native type for strings but provides two ways to handle them: short strings using simple quotes and ByteArray using double quotes.
Short strings
A short string is an ASCII string where each character is encoded on one byte (see the [ASCII table][ascii]). For example:
- `'a'` is equivalent to `0x61`
- `'b'` is equivalent to `0x62`
- `'c'` is equivalent to `0x63`
- `0x616263` is equivalent to `'abc'`.
Cairo uses the `felt252` for short strings. As the `felt252` is on 251 bits, a short string is limited to 31 characters (31 * 8 = 248 bits, which is the maximum multiple of 8 that fits

in 251 bits).
You can choose to represent your short string with an hexadecimal value like
`0x616263` or by directly writing the string using simple quotes like `'abc'`, which is
more convenient.
Here are some examples of declaring short strings in Cairo:
```cairo
{{#rustdoc_include ../listings/ch02-common-programming-concepts/
no_listing_10_short_string_type/src/lib.cairo:2:6}}
```
[ascii]: https://www.asciitable.com/
Byte Array Strings
With the `ByteArray` struct added in Cairo 2.4.0, you are not limited to 31 characters
anymore. These `ByteArray` strings are written in double quotes like in the following
example:
```cairo
{{#rustdoc_include ../listings/ch02-common-programming-concepts/
no_listing_10_short_string_type/src/lib.cairo:8:8}}
```
Compound Types
The Tuple Type
A _tuple_ is a general way of grouping together a number of values with a
variety of types into one compound type. Tuples have a fixed length: once
declared, they cannot grow or shrink in size.
We create a tuple by writing a comma-separated list of values inside
parentheses. Each position in the tuple has a type, and the types of the
different values in the tuple don't have to be the same. We've added optional
type annotations in this example:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_12_tuple_type/src/
lib.cairo}}
```
The variable `tup` binds to the entire tuple because a tuple is considered a
single compound element. To get the individual values out of a tuple, we can
use pattern matching to destructure a tuple value, like this:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_13_tuple_destructuration/src/lib.cairo}}
```
This program first creates a tuple and binds it to the variable `tup`. It then
uses a pattern with `let` to take `tup` and turn it into three separate
variables, `x`, `y`, and `z`. This is called _destructuring_ because it breaks
the single tuple into three parts. Finally, the program prints `y is 6!` as the value of
`y` is `6`.
We can also declare the tuple with value and types, and destructure it at the same time.
For example:
```cairo
```

{{#include ../listings/ch02-common-programming-concepts/no_listing_14_tuple_types/
src/lib.cairo}}
~~~
#### The Unit Type ()
A _unit type_ is a type which has only one value `()`.
It is represented by a tuple with no elements.
Its size is always zero, and it is guaranteed to not exist in the compiled code.
You might be wondering why you would even need a unit type? In Cairo, everything is
an expression, and an expression that returns nothing actually returns `()` implicitly.
### The Fixed Size Array Type
Another way to have a collection of multiple values is with a _fixed size array_. Unlike a
tuple, every element of a fixed size array must have the same type.
We write the values in a fixed-size array as a comma-separated list inside square
brackets. The array's type is written using square brackets with the type of each
element, a semicolon, and then the number of elements in the array, like so:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_40_fixed_size_arr_type/src/lib.cairo}}
~~~
In the type annotation `[u64; 5]`, `u64` specifies the type of each element, while `5`
after the semicolon defines the array's length. This syntax ensures that the array always
contains exactly 5 elements of type `u64`.
Fixed size arrays are useful when you want to hardcode a potentially long sequence of
data directly in your program. This type of array must not be confused with the
[`Array<T>` type][arrays], which is a similar collection type provided by the core library
that _is_ allowed to grow in size. If you're unsure whether to use a fixed size array or
the `Array<T>` type, chances are that you are looking for the `Array<T>` type.
Because their size is known at compile-time, fixed-size arrays don't require runtime
memory management, which makes them more efficient than dynamically-sized arrays.
Overall, they're more useful when you know the number of elements will not need to
change. For example, they can be used to efficiently store lookup tables that won't
change during runtime. If you were using the names of the month in a program, you
would probably use a fixed size array rather than an `Array<T>` because you know it
will always contain 12 elements:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_41_fixed_size_arr_months/src/lib.cairo:months}}
```
You can also initialize an array to contain the same value for each element by
specifying the initial value, followed by a semicolon, and then the length of the array in
square brackets, as shown here:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_41_fixed_size_arr_months/src/lib.cairo:repeated_values}}
```
The array named `a` will contain `5` elements that will all be set to the value `3` initially.

This is the same as writing `let a = [3, 3, 3, 3, 3];` but in a more concise way.
Accessing Fixed Size Arrays Elements
As a fixed-size array is a data structure known at compile time, it's content is represented as a sequence of values in the program bytecode. Accessing an element of that array will simply read that value from the program bytecode efficiently.
We have two different ways of accessing fixed size array elements:
- Deconstructing the array into multiple variables, as we did with tuples.
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_42_fixed_size_arr_accessing_elements/src/lib.cairo}}
```
- Converting the array to a [Span][span], that supports indexing. This operation is _free_ and doesn't incur any runtime cost.
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_44_fixed_size_arr_accessing_elements_span/src/lib.cairo}}
```
Note that if we plan to repeatedly access the array, then it makes sense to call `.span()` only once and keep it available throughout the accesses.
Type Conversion
Cairo addresses conversion between types by using the `try_into` and `into` methods provided by the `TryInto` and `Into` traits from the core library. There are numerous implementations of these traits within the standard library for conversion between types, and they can be implemented for [custom types as well][custom-type-conversion].
Into
The `Into` trait allows for a type to define how to convert itself into another type. It can be used for type conversion when success is guaranteed, such as when the source type is smaller than the destination type.
To perform the conversion, call `var.into()` on the source value to convert it to another type. The new variable's type must be explicitly defined, as demonstrated in the example below.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_11_into/src/
lib.cairo}}
```
TryInto
Similar to `Into`, `TryInto` is a generic trait for converting between types. Unlike `Into`, the `TryInto` trait is used for fallible conversions, and as such, returns [Option\<T\>][option]. An example of a fallible conversion is when the target type might not fit the source value.
Also similar to `Into` is the process to perform the conversion; just call `var.try_into()` on the source value to convert it to another type. The new variable's type also must be explicitly defined, as demonstrated in the example below.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_39_tryinto/src/
lib.cairo}}
```

```

{{#quiz ../quizzes/ch02-02-data-types.toml}}
[arrays]: ./ch03-01-arrays.md
[option]: ./ch06-01-enums.md#the-option-enum-and-its-advantages
[custom-type-conversion]: ./ch05-02-an-example-program-using-structs.md#conversions-of-custom-types
[span]: ./ch03-01-arrays.md#Span
```