Printing
When writing a program, it is quite common to print some data to the console, either for the normal process of the program or for debugging purpose. In this chapter, we describe the options you have to print simple and complex data types.
Printing Standard Data Types
Cairo provides two macros to print standard data types:
- `println!` which prints on a new line
- `print!` with inline printing
Both take a `ByteArray` string as first parameter (see [Data Types][byte array]), which can be a simple string to print a message or a string with placeholders to format the way values are printed.
There are two ways to use these placeholders and both can be mixed:
- empty curly brackets `{}` are replaced by values given as parameters to the `print!` macro, in the same order.
- curly brackets with variable names are directly replaced by the variable value.
Here are some examples:
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_08_print_macro/src/lib.cairo}}
```
> `print!` and `println!` macros use the `Display` trait under the hood, and are therefore used to print the value of types that implement it. This is the case for basic data types, but not for more complex ones. If you try to print complex data type values with these macros, e.g. for debugging purposes, you will get an error. In that case, you can either [manually implement][print with display] the `Display` trait for your type or use the `Debug` trait (see [below][print with debug]).
[byte array]: ./ch02-02-data-types.md#byte-array-strings
[print with display]: ./ch11-08-printing.md#printing-custom-data-types
[print with debug]: ./ch11-08-printing.md#print-debug-traces
Formatting
Cairo also provides a useful macro to handle string formatting: `format!`. This macro works like `println!`, but instead of printing the output to the screen, it returns a `ByteArray` with the contents. In the following example, we perform string concatenation using either the `+` operator or the
`format!` macro. The version of the code using `format!` is much easier to read, and the code generated by the `format!` macro uses snapshots, so that this call doesn't take ownership of any of its parameters.
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_06_format_macro/src/lib.cairo}}
```
Printing Custom Data Types
As previously explained, if you try to print the value of a custom data type with `print!` or `println!` macros, you'll get an error telling you that the `Display` trait is not implemented for your custom type:
```shell
error: Trait has no implementation in context:
core::fmt::Display::<package_name::struct_name>
```

```
The `println!` macro can do many kinds of formatting, and by default, the curly brackets tell `println!` to use formatting known as `Display` - output intended for direct end user consumption. The primitive types we've seen so far implement `Display` by default because there's only one way you'd want to show a `1` or any other primitive type to a user. But with structs, the way `println!` should format the output is less clear because there are more display possibilities: Do we want commas or not? Do we want to print the curly brackets? Should all the fields be shown? Due to this ambiguity, Cairo doesn't try to guess what we want, and structs don't have a provided implementation of `Display` to use with `println!` and the `{}` placeholder.
Here is the `Display` trait to implement:
```cairo,noplayground
trait Display<T> {
    fn fmt(self: @T, ref f: Formatter) -> Result<(), Error>;
}
```
The second parameter `f` is of type `Formatter`, which is just a struct containing a `ByteArray`, representing the pending result of formatting:
```cairo,noplayground
#[derive(Default, Drop)]
pub struct Formatter {
    /// The pending result of formatting.
    pub buffer: ByteArray,
}
```
Knowing this, here is an example of how to implement the `Display` trait for a custom `Point` struct:
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_09_display_trait_with_format/src/lib.cairo}}
```
Cairo also provides the `write!` and `writeln!` macros to write formatted strings in a formatter.
Here is a short example using `write!` macro to concatenate multiple strings on the same line and then print the result:
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_07_write_macro/src/lib.cairo}}
```
It is also possible to implement the `Display` trait for the `Point` struct using these macros, as shown here:
```cairo
{{#include ../listings/ch11-advanced-features/no_listing_10_display_trait_with_write/src/lib.cairo}}
```
> Printing complex data types this way might not be ideal as it requires additional steps to use the `print!` and `println!` macros. If you need to print complex data types,
```

especially when debugging, use the `Debug` trait described below instead.
Print Debug Traces
Cairo provides the `Debug` trait, which can be derived to print the value of variables when debugging. Simply add `:?` within the curly brackets `{}` placeholders in a `print!` or `println!` macro string.
This trait is very useful and is implemented by default for basic data types. It can also be simply derived for complex data types using the `#[derive(Debug)]` attribute, as long as all types they contain implement it. This eliminates the need to manually implement extra code to print complex data types.
Note that `assert_xx!` macros used in tests require the provided values to implement the `Debug` trait, as they also print the result in case of assertion failure.
For more details about the `Debug` trait and its usage for printing values when debugging, please refer to the [Derivable Traits][debug trait] appendix.
[debug trait]: ./appendix-03-derivable-traits.md#debug-trait-for-printing-and-debugging