An Example Program Using Structs

To understand when we might want to use structs, let's write a program that calculates the area of a rectangle. We'll start by using single variables, and then refactor the program until we're using structs instead.

Let's make a new project with Scarb called _rectangles_ that will take the width and height of a rectangle specified in pixels and calculate the area of the rectangle. Listing {{#ref area-fn}} shows a short program with one way of doing exactly that in our project's _src/lib.cairo_.

Filename: src/lib.cairo

```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_03_no_struct/src/lib.cairo:all}}
```

{{#label area-fn}}
Listing {{#ref area-fn}}: Calculating the area of a rectangle specified by separate width and height variables.

Now run the program with `scarb cairo-run`:

```shell
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_03_no_struct/output.txt}}
```

This code succeeds in figuring out the area of the rectangle by calling the `area` function with each dimension, but we can do more to make this code clear and readable.

The issue with this code is evident in the signature of `area`:

```cairo,noplayground
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_03_no_struct/src/lib.cairo:here}}
```

The `area` function is supposed to calculate the area of one rectangle, but the function we wrote has two parameters, and it's not clear anywhere in our program that the parameters are related. It would be more readable and more manageable to group width and height together. We've already discussed one way we might do that in the [Tuple Section of Chapter 2](./ch02-02-data-types.md#the-tuple-type).

Refactoring with Tuples

Listing {{#ref rectangle-tuple}} shows another version of our program that uses tuples.

Filename: src/lib.cairo

```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_04_w_tuples/src/lib.cairo}}
```

{{#label rectangle-tuple}}
Listing {{#ref rectangle-tuple}}: Specifying the width and height of the rectangle with a tuple.

In one way, this program is better. Tuples let us add a bit of structure, and we're now passing just one argument. But in another way, this version is less clear: tuples don't

name their elements, so we have to index into the parts of the tuple, making our calculation less obvious.
Mixing up the width and height wouldn’t matter for the area calculation, but if we want to calculate the difference, it would matter! We would have to keep in mind that `width` is the tuple index `0` and `height` is the tuple index `1`. This would be even harder for someone else to figure out and keep in mind if they were to use our code. Because we haven’t conveyed the meaning of our data in our code, it’s now easier to introduce errors.
Refactoring with Structs: Adding More Meaning
We use structs to add meaning by labeling the data. We can transform the tuple we’re using into a struct with a name for the whole as well as names for the parts.
Filename: src/lib.cairo
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_05_w_structs/src/lib.cairo}}
```
{{#label rectangle-struct}}
Listing {{#ref rectangle-struct}}: Defining a `Rectangle` struct.
Here we’ve defined a struct and named it `Rectangle`. Inside the curly brackets, we defined the fields as `width` and `height`, both of which have type `u64`. Then, in `main`, we created a particular instance of `Rectangle` that has a width of `30` and a height of `10`. Our `area` function is now defined with one parameter, which we’ve named `rectangle` which is of type `Rectangle` struct. We can then access the fields of the instance with dot notation, and it gives descriptive names to the values rather than using the tuple index values of `0` and `1`.
Conversions of Custom Types
We've already described how to perform type conversion on in-built types, see [Data Types > Type Conversion][type-conversion]. In this section, we will see how to define conversions for custom types.
> Note: conversion can be defined for compound types, e.g. tuples, too.
[type-conversion]: ./ch02-02-data-types.md#type-conversion
Into
Defining a conversion for a custom type using the `Into` trait will typically require specification of the type to convert into, as the compiler is unable to determine this most of the time. However this is a small trade-off considering we get the functionality for free.
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_07_into/src/lib.cairo}}
```
TryInto
Defining a conversion for `TryInto` is similar to defining it for `Into`.
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/no_listing_08_tryinto/src/lib.cairo}}
```

{{#quiz ../quizzes/ch05-02-an-example-program-using-structs.toml}}