Defining and Instantiating Structs
Structs are similar to tuples, discussed in the [Data Types](ch02-02-data-types.md) section, in that both hold multiple related values. Like tuples, the pieces of a struct can be different types. Unlike with tuples, in a struct you'll name each piece of data so it's clear what the values mean. Adding these names means that structs are more flexible than tuples: you don't have to rely on the order of the data to specify or access the values of an instance.
To define a struct, we enter the keyword `struct` and name the entire struct. A struct's name should describe the significance of the pieces of data being grouped together. Then, inside curly brackets, we define the names and types of the pieces of data, which we call fields. For example, Listing {{#ref user-struct}} shows a struct that stores information about a user account.
<span class="filename">Filename: src/lib.cairo</span>
```cairo, noplayground
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_01_user_struct/
src/lib.cairo:user}}
```
{{#label user-struct}}
<span class="caption">Listing {{#ref user-struct}}: A `User` struct definition</span>
To use a struct after we've defined it, we create an _instance_ of that struct by specifying concrete values for each of the fields.
We create an instance by stating the name of the struct and then add curly brackets containing _key: value_ pairs, where the keys are the names of the fields and the values are the data we want to store in those fields. We don't have to specify the fields in the same order in which we declared them in the struct. In other words, the struct definition is like a general template for the type, and instances fill in that template with particular data to create values of the type.
For example, we can declare two particular users as shown in Listing {{#ref user-instances}}.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#include ../listings/ch05-using-structs-to-structure-related-data/listing_01_user_struct/
src/lib.cairo:all}}
```
{{#label user-instances}}
<span class="caption">Listing {{#ref user-instances}}: Creating two instances of the `User` struct</span>
To get a specific value from a struct, we use dot notation. For example, to access `user1`'s email address, we use `user1.email`. If the instance is mutable, we can change a value by using the dot notation and assigning into a particular field. Listing {{#ref user-mut}} shows how to change the value in the `email` field of a mutable `User` instance.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#rustdoc_include ../listings/ch05-using-structs-to-structure-related-data/
listing_02_mut_struct/src/lib.cairo:main}}
```

~~~
{{#label user-mut}}
<span class="caption">Listing {{#ref user-mut}}: Changing the value in the email field of a `User` instance</span>
Note that the entire instance must be mutable; Cairo doesn’t allow us to mark only certain fields as mutable.
As with any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return that new instance.
Listing {{#ref build-user}} shows a `build_user` function that returns a `User` instance with the given email and username. The `active` field gets the value of `true`, and the `sign_in_count` gets a value of `1`.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#rustdoc_include ../listings/ch05-using-structs-to-structure-related-data/listing_02_mut_struct/src/lib.cairo:build_user}}
~~~
{{#label build-user}}
<span class="caption">Listing {{#ref build-user}}: A `build_user` function that takes an email and username and returns a `User` instance.</span>
It makes sense to name the function parameters with the same name as the struct fields, but having to repeat the `email` and `username` field names and variables is a bit tedious. If the struct had more fields, repeating each name would get even more annoying. Luckily, there’s a convenient shorthand!
Using the Field Init Shorthand
Because the parameter names and the struct field names are exactly the same in Listing {{#ref build-user}}, we can use the field init shorthand syntax to rewrite `build_user` so it behaves exactly the same but doesn’t have the repetition of `username` and `email`, as shown in Listing {{#ref init-shorthand}}.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#rustdoc_include ../listings/ch05-using-structs-to-structure-related-data/listing_02_mut_struct/src/lib.cairo:build_user2}}
```
{{#label init-shorthand}}
<span class="caption">Listing {{#ref init-shorthand}}: A `build_user` function that uses field init shorthand because the `username` and `email` parameters have the same name as struct fields.</span>
Here, we’re creating a new instance of the `User` struct, which has a field named `email`. We want to set the `email` field’s value to the value in the `email` parameter of the `build_user` function. Because the `email` field and the `email` parameter have the same name, we only need to write `email` rather than `email: email`.
Creating Instances from Other Instances with Struct Update Syntax
It’s often useful to create a new instance of a struct that includes most of
the values from another instance, but changes some. You can do this using
struct update syntax.
First, in Listing {{#ref without-update-syntax}} we show how to create a new `User`

instance in `user2`
regularly, without the update syntax. We set a new value for `email` but
otherwise use the same values from `user1` that we created in Listing {{#ref user-instances}}.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#rustdoc_include ../listings/ch05-using-structs-to-structure-related-data/listing_without_update_syntax/src/lib.cairo:here}}
```
{{#label without-update-syntax}}
<span class="caption">Listing {{#ref without-update-syntax}}: Creating a new `User` instance using all but one of the values from `user1`</span>
Using struct update syntax, we can achieve the same effect with less code, as
shown in Listing {{#ref update-syntax}}. The syntax `..` specifies that the remaining fields not
explicitly set should have the same value as the fields in the given instance.
<span class="filename">Filename: src/lib.cairo</span>
```cairo
{{#rustdoc_include ../listings/ch05-using-structs-to-structure-related-data/listing_update_syntax/src/lib.cairo:here}}
```
{{#label update-syntax}}
<span class="caption">Listing {{#ref update-syntax}}: Using struct update syntax to set a new
`email` value for a `User` instance but to use the rest of the values from `user1`</span>
The code in Listing {{#ref update-syntax}} also creates an instance of `user2` that has a
different value for `email` but has the same values for the `username`,
`active`, and `sign_in_count` fields as `user1`. The `..user1` part must come last
to specify that any remaining fields should get their values from the
corresponding fields in `user1`, but we can choose to specify values for as
many fields as we want in any order, regardless of the order of the fields in
the struct’s definition.
Note that the struct update syntax uses `=` like an assignment; this is because it moves the data,
just as we saw in the ["Moving Values"][move]<!-- ignore --> section. In this example, we can no
longer use `user1` as a whole after creating `user2` because the `ByteArray` in the
`username` field of `user1` was moved into `user2`. If we had given `user2` new
`ByteArray` values for both `email` and `username`, and thus only used the
`active` and `sign_in_count` values from `user1`, then `user1` would still be
valid after creating `user2`. Both `active` and `sign_in_count` are types that
implement the `Copy` trait, so the behavior we discussed in the ["`Copy` Trait"][copy]<!-- ignore --> section would apply.
{{#quiz ../quizzes/ch05-01-defining-and-instantiating-structs.toml}}
[move]: ch04-01-what-is-ownership.md#moving-values
[copy]: ch04-01-what-is-ownership.md#the-copy-trait