The Move Book

Move's type system shines when it comes to defining custom types. User defined types can be custom tailored to the specific needs of the application, not only on the data level, but also in its behavior. In this section we introduce the struct definition and how to use it.

To define a custom type, you can use the struct keyword followed by the name of the type. After the name, you can define the fields of the struct. Each field is defined with the field_name: field_type syntax. Field definitions must be separated by commas. The fields can be of any type, including other structs.

Move does not support recursive structs, meaning a struct cannot contain itself as a field.

In the example above, we define a Record struct with five fields. The title field is of type String , the artist field is of type Artist , the year field is of type u16 , the is_debut field is of type bool , and the edition field is of type Option . The edition field is of type Option to represent that the edition is optional.

Structs are private by default, meaning they cannot be imported and used outside of the module they are defined in. Their fields are also private and can't be accessed from outside the module. See visibility for more information on different visibility modifiers.

Fields of a struct are private and can only be accessed by the module defining the struct. Reading and writing the fields of a struct in other modules is only possible if the module defining the struct provides public functions to access the fields.

We described the definition of a struct. Now let's see how to initialize a struct and use it. A struct can be initialized using the struct_name { field1: value1, field2: value2, ... } syntax. The fields can be initialized in any order, and all of the required fields must be set.

In the example above, we create an instance of the Artist struct and set the name field to a string "The Beatles".

To access the fields of a struct, you can use the . operator followed by the field name.

Only the module defining the struct can access its fields (both mutably and immutably). So the above code should be in the same module as the Artist struct.

Structs are non-discardable by default, meaning that the initialized struct value must be used, either by storing it or unpacking it. Unpacking a struct means deconstructing it into its fields. This is done using the let keyword followed by the struct name and the field names.

In the example above we unpack the Artist struct and create a new variable name with the value of the name field. Because the variable is not used, the compiler will raise a warning. To suppress the warning, you can use the underscore _ to indicate that the variable is intentionally unused.

Struct

To define a custom type, you can use the struct keyword followed by the name of the type. After the name, you can define the fields of the struct. Each field is defined with the field_name: field_type syntax. Field definitions must be separated by commas. The fields can be of any type, including other structs.

Move does not support recursive structs, meaning a struct cannot contain itself as a field.

```bash /// A struct representing an artist. public struct Artist { /// The name of the artist. name: String, }

/// A struct representing a music record. public struct Record { /// The title of the record. title: String, /// The artist of the record. Uses the `Artist` type. artist: Artist, /// The year the record was released. year: u16, /// Whether the record is a debut album. is_debut: bool, /// The edition of the record. edition: Option, } ```

In the example above, we define a Record struct with five fields. The title field is of type String , the artist field is of type Artist , the year field is of type u16 , the is_debut field is of type bool , and the edition field is of type Option . The edition field is of type Option to represent that the edition is optional.

Structs are private by default, meaning they cannot be imported and used outside of the module they are defined in. Their fields are also private and can't be accessed from outside the module. See visibility for more information on different visibility modifiers.

Fields of a struct are private and can only be accessed by the module defining the struct. Reading and writing the fields of a struct in
```

other modules is only possible if the module defining the struct provides public functions to access the fields.

We described the definition of a struct. Now let's see how to initialize a struct and use it. A struct can be initialized using the struct_name { field1: value1, field2: value2, ... } syntax. The fields can be initialized in any order, and all of the required fields must be set.

```
bash let mut artist = Artist { name: b"The Beatles".to_string() };
```

In the example above, we create an instance of the Artist struct and set the name field to a string "The Beatles".

To access the fields of a struct, you can use the . operator followed by the field name.

``bash // Access thenamefield of theArtist` struct. let artist_name = artist.name;

// Access a field of the `Artist` struct. assert!(artist.name == b"The Beatles".to_string());

// Mutate the `name` field of the `Artist` struct. artist.name = b"Led Zeppelin".to_string();

// Check that the `name` field has been mutated. assert!(artist.name == b"Led Zeppelin".to_string()); ```

Only the module defining the struct can access its fields (both mutably and immutably). So the above code should be in the same module as the Artist struct.

Structs are non-discardable by default, meaning that the initialized struct value must be used, either by storing it or unpacking it. Unpacking a struct means deconstructing it into its fields. This is done using the let keyword followed by the struct name and the field names.

```
bash // Unpack the `Artist` struct and create a new variable `name` // with the value of the `name` field. let Artist { name } = artist;
```

In the example above we unpack the Artist struct and create a new variable name with the value of the name field. Because the variable is not used, the compiler will raise a warning. To suppress the warning, you can use the underscore _ to indicate that the variable is intentionally unused.

```
bash // Unpack the `Artist` struct and ignore the `name` field. let Artist { name: _ } = artist;
```

Create and use an instance

We described the definition of a struct. Now let's see how to initialize a struct and use it. A struct can be initialized using the struct_name { field1: value1, field2: value2, ... } syntax. The fields can be initialized in any order, and all of the required fields must be set.

```
bash let mut artist = Artist { name: b"The Beatles".to_string() };
```

In the example above, we create an instance of the Artist struct and set the name field to a string "The Beatles".

To access the fields of a struct, you can use the . operator followed by the field name.

``bash // Access thenamefield of theArtist` struct. let artist_name = artist.name;

// Access a field of the `Artist` struct. assert!(artist.name == b"The Beatles".to_string());

// Mutate the `name` field of the `Artist` struct. artist.name = b"Led Zeppelin".to_string();

// Check that the `name` field has been mutated. assert!(artist.name == b"Led Zeppelin".to_string()); ```

Only the module defining the struct can access its fields (both mutably and immutably). So the above code should be in the same module as the Artist struct.

Structs are non-discardable by default, meaning that the initialized struct value must be used, either by storing it or unpacking it. Unpacking a struct means deconstructing it into its fields. This is done using the let keyword followed by the struct name and the field names.

```
bash // Unpack the `Artist` struct and create a new variable `name` // with the value of the `name` field. let Artist { name } = artist;
```

In the example above we unpack the Artist struct and create a new variable name with the value of the name field. Because the

variable is not used, the compiler will raise a warning. To suppress the warning, you can use the underscore _ to indicate that the variable is intentionally unused.

```
bash // Unpack the `Artist` struct and ignore the `name` field. let Artist { name: _ } = artist;
```

Unpacking a struct

Structs are non-discardable by default, meaning that the initialized struct value must be used, either by storing it or unpacking it. Unpacking a struct means deconstructing it into its fields. This is done using the let keyword followed by the struct name and the field names.

```
bash // Unpack the `Artist` struct and create a new variable `name` // with the value of the `name` field. let Artist { name } = artist;
```

In the example above we unpack the Artist struct and create a new variable name with the value of the name field. Because the variable is not used, the compiler will raise a warning. To suppress the warning, you can use the underscore _ to indicate that the variable is intentionally unused.

```
bash // Unpack the `Artist` struct and ignore the `name` field. let Artist { name: _ } = artist;
```

Further reading