Structs and enums

Overview

Structs

If you're not familiar with Rust, it may be confusing that there are no classes or inheritance like other programming languages. We'll be exploring how touse structs , which are someone similar to classes, but perhaps simpler.

Remember that there will be only one struct that gets the#[near_bindgen] macro placed on it; our primary struct or singleton if you wish. Oftentimes the primary struct will contain additional structs that may, in turn, contain more structs in a neat and orderly way. You may also have structs that are used to return data to an end user, like a frontend. We'll be covering both of these cases in this chapter.

Enums

Enums are short for enumerations, and can be particularly useful if you have entities in your smart contract that transition to different states. For example, say you have a series of blockchain games where players can join, battle, and win. There might be an enumeration forAcceptingPlayers ,GameInProgress , andGameCompleted . Enums are also used to define discrete types of concept, like months in a year.

For our crossword puzzle, one example of an enum is the direction of the clue: either across (A) or down (D) as illustrated below. These are the only two options.

Art byeizaconiendo.near

Rust has an interesting feature where enums can contain additional data. You can seeexamples of that here .

Using structs

Storing contract state

We're going to introduce several structs all at once. These structs are addressing a need from the previous chapter, where the puzzle itself was hardcoded and looked like this:

In this chapter, we want the ability to add multiple, custom crossword puzzles. This means we'll be storing information about the clues in the contract state. Think of a grid where there are x and y coordinates for where a clue starts. We'll also want to specify:

1. Clue number
2. Whether it'sacross
3. ordown
4. The length, or number of letters in the answer

Let's dive right in, starting with our primary struct:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

pub

struct

Crossword

{ puzzles :

LookupMap < String ,

Puzzle

,

// ⟵ Puzzle is a struct we're defining unsolved_puzzles :

UnorderedSet < String

, } Let's ignore a couple of things… For now, let's ignore the macros about the structs that begin withderive andserde . Look at the fields inside theCrossword struct above, and you'll see a couple types.String is a part of Rust's standard library, butPuzzle is something we've created:

[derive(BorshDeserialize, BorshSerialize, Debug)]

pub

struct

Puzzle

{ status :

PuzzleStatus ,

// ⟵ An enum we'll get to soon /// Use the CoordinatePair assuming the origin is (0, 0) in the top left side of the puzzle. answer :

Vec < Answer

,

// ⟵ Another struct we've defined } Let's focus on theanswer field here, which is a vector ofAnswer s. (A vector is nothing fancy, just a bunch of items or a "growable array" as described in thestandard Rust documentation .

[derive(BorshDeserialize, BorshSerialize, Deserialize, Serialize, Debug)]

[serde(crate =

"near_sdk::serde")] pub

struct

Answer

{ num :

u8 , start :

CoordinatePair ,

// ⟵ Another struct we've defined direction :

AnswerDirection ,

// ⟵ An enum we'll get to soon length :

u8 , clue :

String , } Now let's take a look at the last struct we'e defined, that has cascaded down from fields on our primary struct: theCoordinatePair .

[derive(BorshDeserialize, BorshSerialize, Deserialize, Serialize, Debug)]

[serde(crate =

"near_sdk::serde")] pub

struct

CoordinatePair

{ x :

u8 , y :

u8 , } Summary of the structs shown There are a handful of structs here, and this will be a typical pattern when we use structs to store contract state.

Crossword ⟵ primary struct with #[near_bindgen] └─── Puzzle └─── Answer └─── CoordinatePair

Returning data

Since we're going to have multiple crossword puzzles that have their own, unique clues and positions in a grid, we'll want to return puzzle objects to a frontend.

Quick note on return values By default, return values are serialized in JSON unless explicitly directed to use Borsh for binary serialization.

For example, if we call this function:

pub

fn

return_some_words ( )

->

Vec < String

{ vec! ["crossword" . to_string ( ) ,

"puzzle" . to_string ( )] } The return value would be a JSON array:

["crossword", "puzzle"]

While somewhat advanced, you can learn more aboutchanging the serialization here . We have a struct calledJsonPuzzle that differs from thePuzzle struct we've shown. It has one difference: the addition of thesolution_hash field.

[derive(Serialize, Deserialize)]

[serde(crate =

"near_sdk::serde")] pub

struct

JsonPuzzle

{ /// The human-readable (not in bytes) hash of the solution solution_hash :

String ,

// ⟵ this field is not contained in the Puzzle struct status :

PuzzleStatus , answer :

Vec < Answer

, } This is handy because our primary struct has a key-value pair where the key is the solution hash (as aString) and the value is thePuzzle struct.

pub

struct

Crossword

{ puzzles :

LookupMap < String ,

Puzzle

, // key ↗↖ value … OurJsonPuzzle struct returns the information from both the key and the value.

We can move on from this topic, but suffice it to say, sometimes it's helpful to have structs where the intended use is to return data in a more meaningful way than might exist from the structs used to store contract data.

Using returned objects in a callback

Don't be alarmed if this section feels confusing at this point, but know we'll cover Promises and callbacks later.

Without getting into detail, a contract may want to make a cross-contract call and "do something" with the return value. Sometimes this return value is an object we're expecting, so we can define a struct with the expected fields to capture the value. In other programming languages this may be referred to as "casting" or "marshaling" the value.

A real-world example of this might be theStorage Management standard , as used in afungible token .

Let's say a smart contract wants to determine ifalice.near is "registered" on thenDAI token. More technically, doesalice.near have a key-value pair for herself in the fungible token contract.

[derive(Serialize, Deserialize)]

[serde(crate =

"near_sdk::serde")] pub

struct

StorageBalance

{ pub total :

U128 , pub available :

U128 , }

// … // Logic that calls the nDAI token contract, asking for alice.near's storage balance. // …

[private]

pub

fn

my_callback (& mut

self ,

[callback]

storage_balance :

StorageBalance)

{ // … } The crossword puzzle will eventually use a cross-contract call and callback, so we can look forward to that. For now just know that if your contract expects to receive a return value that's not a primitive (unsigned integer, string, etc.) and is more complex, you may use a struct to give it the proper type.

Using enums

In the section above, we saw two fields in the structs that had an enum type:

1.AnswerDirection — this is the simplest type of enum, and will look familiar from other programming languages. It provides the only two options for how a clue in oriented in a crossword puzzle: across and down.

[derive(BorshDeserialize, BorshSerialize, Deserialize, Serialize, Debug)]

[serde(crate =

"near_sdk::serde")] pub

enum

AnswerDirection

{ Across , Down , } 1. PuzzleStatus 2. — this enum can actually store a string inside theSolved 3. structure. (Note that we could have simply stored a string instead of having a structure, but a structure might make this easier to read.)

As we improve our crossword puzzle, the idea is to give the winner of the crossword puzzle (the first person to solve it) the ability to write a memo. (For example: "Took me forever to get clue six!", "Alice rules!" or whatever.)

[derive(BorshDeserialize, BorshSerialize, Deserialize, Serialize, Debug)]

[serde(crate =

"near_sdk::serde")] pub

enum

PuzzleStatus

{ Unsolved , Solved

{ memo :

String

} , } Edit this page Last updatedonAug 25, 2022 byDamián Parrino Was this page helpful? Yes No

Previous Store multiple puzzles Next Actions and sending NEAR