Structs

Structs

Copy page

Practice using structs.

The `struct` type allows you to organize related data of different types.

Objectives

By the end of this lesson you should be able to:

- Construct a `struct` (user-defined type) that contains several different data types
- Declare members of the `struct` to maximize storage efficiency
- Describe constraints related to the assignment of `struct` s depending on the types they contain

Creating a Struct

In the last exercise, we used a `mapping` to create a relationship between an `address` and a `uint`. But what if your users have favorite colors too? Or favorite cars? You **could** create a `mapping` for each of these, but it would quickly get awkward. Instead, a `struct` can be used to create a custom type that can store all of a user's favorites within one data type.

Create a new contract called `Structs`.

Setting up the Struct

Instantiate a `struct` with the keyword, followed by a name for the type, curly brackets, and the variables that make up the type. Add a stub for `Favorites`:

```
struct Favorites {

}
```

After consulting with the designers, we need to store the following for each address's favorites:

- Favorite number
- Birth Day of Month
- Favorite color

Lucky Lottery numbers

Let's pause for a moment and do some technical design around how to save our favorites.

The product team has confirmed for us that we can safely expect that no users have a favorite number greater than 65,536, and of course, everyone is born on a day of the month between 1-31.

Variable **packing** also works inside structs, so we could potentially save on storage by using smaller `uint` s for those variables. However, people don't change their favorite number very often, and the day of the month that they were born on never changes.

Therefore, it's probably more gas-efficient and less cumbersome to write other parts of the code, if we just use `uint` for both variables.

Favorite color can be a `string` .

For Lucky Lottery Numbers, we need a collection. We could use a dynamic array, since this will be in *storage*, but we already know that the lottery has 5 numbers.

Try to use this information to build the `struct` on your own. You should end up with something similar to:

Reveal code

Instantiating a Struct with Its Name

There are two ways to instantiate a struct using its name. The first is similar to instantiating a new object in JavaScript:

```
Favorites memory myFavorites = Favorites({
    favoriteNumber: 29,
    birthDay: 14,
    favoriteColor: "red",
    lotteryNumbers: [uint(1), 2, 3, 4, 5]
});
```

You can also use a shorthand method where you skip the member names and just list a value for each one. Note that the curly brackets are **not** included in this format:

```
Favorites memory myFavorites = Favorites(
    29,
    14,
    "red",
    [uint(1), 2, 3, 4, 5]
);
```

There's no difference in gas costs with either of these methods. Use the one that makes the most sense for the given situation.

Saving Multiple Instances to Storage

Next, we need to figure out the best way to organize the `Favorites` in *storage*. There are a few options, as always, each with tradeoffs. You could match the pattern you used for favorite numbers and utilize a `mapping` to match `addresses` to `Favorites`.

Another popular method is to use an array, which takes advantage of `.push` returning a reference to the newly added element, and the fact that the concept of *undefined* does not exist in Solidity.

First, instantiate an array of `Favorites`:

```
Favorites[] public userFavorites;
```

Next, add a `public` function to add submitted favorites to the list. It should take each of the members as an argument. Then, assign each argument to the new element via the reference returned by `push()`.

Reveal code

Alternatively, you can create an instance in memory, then `push` it to storage.

Reveal code

The gas cost is similar for each of these methods.

Unexpected Behavior in Structs

Structs in Solidity exhibit some properties that are unexpected, or even frustrating. Working with them often includes untangling a set of mutually-exclusive properties and needs.

Dynamic Storage Arrays in Structs

The product team has contacted you to let you know that the beta testers are complaining about the `lotteryNumbers`. As it turns out, not every locality has lotteries where 5 numbers are drawn. Some have 3, 4, or even 6!.

You might think this is an easy enough change. After all, you can just remove the size from the array declaration inside `Favorites`. Go ahead and try it:

```
struct Favorites {
    uint favoriteNumber;
    uint birthDay;
    string favoriteColor;
    uint[] lotteryNumbers; // Removed the '5'
}
```

You'll get an error if you're using the `memory` method shown above.

```
from solidity:
TypeError: Invalid type for argument in function call. Invalid implicit con
  --> contracts/mappings_exercise.sol:70:13:
   |
70 |             [uint(1), 2, 3, 4, 5]
   |             ^^^^^^^^^^^^^^^^^^^^^
```

The simplest resolution here is to switch back to using `push()` to create an empty instance of `Favorites` , then assigning the values.

The reason this works is a little obtuse. In the failing example, an unsized `uint` array is the expected type for the argument, but a sized `uint` array is provided. Solidity cannot perform implicit conversions like this most of the time and you'll get a compiler error if you provide the wrong type for an argument, even if it is convertible.

One exception to this rule is that Solidity **can** perform an implicit conversion during *assignment* if the variable on the right side "fits" into the variable on the left side.

`uint[5]` fits in `uint[]` , so Solidity will allow it to sit 🐈.

But what happens if you use the *getter* for `userFavorites` to retrieve your entry?

```
{
    "0": "uint256: favoriteNumber 29",
    "1": "uint256: birthDay 14",
    "2": "string: favoriteColor red"
}
```

What happened to the array? It’s not there, and it turns out that this is **on purpose**.

Mappings Inside of Structs

You may add `mappings` inside of `struct` s, subject to a few quirks and restrictions. Add `mapping (uint => uint) numberPairs;` to `Favorites` .

In `addFavorites` , assign `newFavorite.numberPairs[33] = 66;`

Deploy and test. So far, so good!

Déjà vu ahead: But what happens if you use the *getter* for `userFavorites` to retrieve your entry?

```
{
    "0": "uint256: favoriteNumber 29",
    "1": "uint256: birthDay 14",
    "2": "string: favoriteColor red"
}
```

It's not there, and it turns out that this is **on purpose**.

Another issue emerges if you try to return the struct from a public function. What if you wanted your `addFavorite` function to return a reference to the new favorite?

```
// Bad code example, will not work
function addFavorite(
    uint _favoriteNumber,
    uint _birthDay,
    string calldata _favoriteColor,
    uint[] calldata _lotteryNumbers
) public returns (newFavorite memory) {
    // .push() returns a reference to the new element
    Favorites storage newFavorite = userFavorites.push();
    newFavorite.favoriteNumber = _favoriteNumber;
    newFavorite.birthDay = _birthDay;
    newFavorite.favoriteColor = _favoriteColor;
    newFavorite.lotteryNumbers = _lotteryNumbers;
    newFavorite.numberPairs[33] = 66;

    return newFavorite;
}
```

You'll get an error. The `mapping` type cannot be returned by a `public` or `external` function, so neither can a `struct` that contains one.

```
from solidity:
TypeError: Types containing (nested) mappings can only be parameters or ret
  --> contracts/mappings_exercise.sol:64:23:
   |
64 |     ) public returns (Favorites memory) {
   |                       ^^^^^^^^^^^^^^^^
```

Finally, what happens if you try to assign `newFavorite` to a `memory` variable? Again, an error occurs because `mapping` s can only be in `storage` .

```
// Bad code example, will not work
Favorites memory secondFavorite = newFavorite;
```

```
from solidity:
TypeError: Type struct Structs.Favorites memory is only valid in storage be
  --> contracts/mappings_exercise.sol:82:9:
   |
82 |         Favorites memory secondFavorite = newFavorite;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Automatic Getters for Public Structs

As with other types, if you put a `public` `struct` in storage at the contract level, the compiler will generate a getter automatically. However, these don't work quite the way you might expect. For example, imagine:

```
struct MyStruct {
    uint first;
    uint second;
    uint third;
}

MyStruct myStruct;
```

```
function myStruct() public view returns (MyStruct memory) {
    return myStruct;
}
```

Instead, it returns the members individually:

```
// Approximate example, not real code
function myStruct() public view returns (uint, uint, uint) {
    return (myStruct.first, myStruct.second, myStruct.third);
}
```

Create your own getter to return the data as a tuple, which will be interpreted as the appropriate type if it's called from another contract via an interface.

```
function getMyStruct() public view returns (MyStruct memory) {
    return myStruct;
}
```

Ask a question... Ctrl+I

Conclusion

In this lesson, you've learned how to use the `struct` keyword to create a custom type that stores related data. You've also learned three methods of instantiating them and common patterns for storing `struct` s in storage. Finally, you've explored some of the constraints that emerge when working with more complex data types within a `struct` .

Was this page helpful? Yes No Suggest edits Raise issue

‹ Structs Exercise ›

base docs

B e.org Blog Privacy Policy Terms of Service Cookie Policy