Contracts and Basic Functions

Basic Types

Copy page

Introduction to basic types in Solidity.

Solidity contains most of the basic **types** you are used to from other languages, but their properties and usage are often a little different than other languages and are likely much more restrictive. In particular, Solidity is a very **explicit** language and will not allow you to make inferences most of the time.

Objectives

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

- Categorize basic data types
- List the major differences between data types in Solidity as compared to other languages

Compare and contrast signed and unsigned integers

Common Properties

In Solidity, **types** must always have a value and are never `undefined`, `null`, or `none`. Because of this, each type has a default value. If you declare a variable without assigning a value, it will instead have the default value for that type. This property can lead to some tricky bugs until you get used to it.

```
uint defaultValue;
uint explicitValue = 0;

// (defaultValue == explicitValue) <-- true
```

Types can be cast from one type to another, but not as freely as you may expect. For example, to convert a `uint256` into a `int8`, you need to cast twice:

```
uint256 first = 1;
int8 second = int8(int256(first));
```

Overflow/underflow protection (described below), does not provide protection when casting.

```
uint256 first = 256;
int8 second = int8(int256(first)); // <- The value stored in second
```

Boolean

Booleans can have a value of `true` or `false`. Solidity does not have the concept of *truthy* or *falsey*, and non-boolean values cannot be cast to bools by design. The short conversation in this **issue** explains why, and explains the philosophy why.

Logical Operators

Standard logical operators (`!`, `&&`, `||`, `==`, `!=`) apply to booleans. Short-circuiting rules do apply, which can sometimes be used for gas savings since if the first operator in an `&&` is `false` or `||` is `true`, the second will not be evaluated. For example, the following code will execute without an error, despite the divide by zero in the second statement.

```
// Bad code for example.  Do not use.
uint divisor = 0;
if(1 < 2 || 1 / divisor > 0) {
    // Do something...
}
```

You cannot use any variant of `>` or `<` with booleans, because they cannot be implicitly or explicitly cast to a type that uses those operators.

Numbers

Solidity has a number of types for signed and unsigned **integers**, which are not ignored as much as they are in other languages, due to potential gas-savings when storing smaller numbers. Support for **fixed point numbers** is under development, but is not fully implemented as of version `0.8.17` .

Floating point numbers are not supported and are not likely to be. Floating precision includes an inherent element of ambiguity that doesn't work for explicit environments like blockchains.

Min, Max, and Overflow

Minimum and maximum values for each type can be accessed with `type(<type>).min` and `type(<type>).max`. For example, `type(uint).min` is **0**, and `type(uint).max` is equal to **2^256-1**.

An overflow or underflow will cause a transaction to *revert*, unless it occurs in a code block that is marked as **unchecked**.

`uint` vs. `int`

In Solidity, it is common practice to favor `uint` over `int` when it is known that a value will never (or should never) be below zero. This practice helps you write more secure code by requiring you to declare whether or not a given value should be allowed to be negative. Use `uint` for values that should not, such as array indexes, account balances, etc. and `int` for a value that does **need** to be negative.

Integer Variants

Smaller and larger variants of integers exist in many languages but have fallen out of favor in many instances, in part because memory and storage are relatively cheap. Solidity supports sizes in steps of eight from `uint8` to `uint256`, and the same for `int`.

Smaller sized integers are used to optimize gas usage in storage operations, but there is a cost. The EVM operates with 256 bit words, so operations involving smaller data types must be cast first, which costs gas.

`uint` is an alias for `uint256` and can be considered the default.

Operators

Comparisons (`<=` , `<` , `==` , `!=` , `>=` , `>`) and arithmetic (`+` , `-` , `*` , `/` , `%` , `**`) operators are present and work as expected. You can also use bit and shift operators.

`uint` and `int` variants can be compared directly, such as `uint8` and `uint256` , but you must cast one value to compare a `uint` to an `int` .

```
uint first = 1;
int8 second = 1;

if(first == uint8(second)) {
    // Do something...
}
```

Addresses

The **address** type is a relatively unique type representing a wallet or contract address. It holds a 20-byte value, similar to the one we explored when you deployed your *Hello World* contract in *Remix*. `address payable` is a variant of `address` that

allows you to use the `transfer` and `send` methods. This distinction helps prevent sending Ether, or other tokens, to a contract that is not designed to receive it. If that were to happen, the Ether would be lost.

Addresses are **not** strings and do not need quotes when represented literally, but conversions from `bytes20` and `uint160` are allowed.

```
address existingWallet = 0xd9145CCE52D386f254917e481eB44e9943F39138;
```

Members of Addresses

Addresses contain a number of functions. `balance` returns the balance of an address, and `transfer`, mentioned above, can be used to send `ether`.

```
function getBalance(address _address) public view returns(uint) {
    return _address.balance;
}
```

Later on, you'll learn about `call`, `delegatecall`, and `staticcall`, which can be used to call functions deployed in other contracts.

Contracts

When you declare a **contract**, you are defining a type. This type can be used to instantiate one contract as a local variable inside a second contract, allowing the second to interact with the first.

Byte Arrays and Strings

Byte arrays come as both fixed-size and dynamically-sized. They hold a sequence of bytes. Arrays are a little more complicated than in other languages and will be covered in-depth later.

Strings

Strings are arrays in Solidity, not a type. You cannot concat them with `+`, but as of *0.8.12*, you can use `string.concat(first, second)`. They are limited to printable characters and escaped characters. Casting other data types to `string` is at best tricky, and sometimes impossible.

Generally speaking, you should be deliberate when working with strings inside of a smart contract. Don't be afraid to use them when appropriate, but if possible, craft and display messages on the front end rather than spending gas to assemble them on the back end.

Enums

Enums allow you to apply human-readable labels to a list of unsigned integers.

```
enum Flavors { Vanilla, Chocolate, Strawberry, Coffee }

Flavors chosenFlavor = Flavors.Coffee;
```

Enums can be explicitly cast to and from `uint`, but not implicitly. They are limited to 256 members.

Constant and Immutable

The **constant and immutable** keywords allow you to declare variables that cannot be changed. Both result in gas savings because the compiler does not need to reserve a storage slot for these values.

As of *0.8.17*, `constant` and `immutable` are not fully implemented. Both are supported on **value types**, and `constant` can also be used with strings.

Constant

Constants can be declared at the file level, or at the contract level. In Solidity, modifiers come after the type declaration. You must initialize a value when declaring a constant. Convention is to use SCREAMING_SNAKE_CASE for constants.

```
uint constant NUMBER_OF_TEAMS = 10;

contract Cars {
```


At compilation, the compiler replaces every instance of the constant variable with its literal value.

Immutable

The immutable keyword is used to declare variables that are set once within the constructor, which are then never changed:

```
contract Season {
    immutable numberOfRaces;

    constructor(uint _numberOfRaces) {
        numberOfRaces = _numberOfRaces;
    }
}
```

Conclusion

You've learned the usage and some of the unique quirks of common variable types in Solidity. You've seen how overflow and underflow are handled and how that behavior can be overridden. You've learned why unsigned integers are used more commonly than in other languages, why floats are not present, and have been introduced to some of the quirks of working with strings. Finally, you've been introduced to the address and contract data types.

Hello World Guide

Basic Functions Exercise