Solidity Types - Booleans

`bool` : The possible values are constants `true` and `false` .

Solidity Types - Booleans

Operators:

- `!` (logical negation)
- `&&` (logical conjunction, “and”)
- `||` (logical disjunction, “or”)
- `==` (equality)
- `!=` (inequality)

Solidity Types - Integers

`int` / `uint` : Signed and unsigned integers of various sizes. Keywords `uint8` to

Solidity Types - Integers

`int` / `uint` : Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Solidity Types - Integers

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: `<<` (left shift), `>>` (right shift)
- Arithmetic operators: `+`, `-`, unary `-`, `*`, `/`, `%` (modulo), `**` (exponentiation)

Solidity Types - Address

The address type comes in two flavours, which are largely identical:

- `address`: Holds a 20 byte value (size of an Ethereum address).
- `address payable`: Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while a plain `address` cannot be sent Ether.

Solidity Types - Address

Operators:

- `<=`, `<`, `==`, `!=`, `>=` and `>`

Solidity Types - Address Members

`balance` and `transfer`

Solidity Types - Address Members

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Solidity Types - Fixed-Size Byte Arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32. `byte` is an alias for `bytes1`.

Solidity Types - Fixed-Size Byte Arrays

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: `<<` (left shift), `>>` (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

Solidity Types - Fixed-Size Byte Arrays Members

`.length` yields the fixed length of the byte array (read-only).

Solidity Types - Dynamically-Sized Byte Array

`bytes`:

Dynamically-sized byte array, see Arrays. Not a value-type!

`string`:

Dynamically-sized UTF-8-encoded string, see Arrays. Not a value-type!

Solidity Types - Function Types

```
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return types>)]
```

Solidity Types - Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

Solidity Types - Array Members

length:

Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created. For dynamically-sized arrays (only available for storage), this member can be assigned to resize the array. Accessing elements outside the current length does not automatically resize the array and instead causes a failing assertion. Increasing the length adds new zero-initialised elements to the array. Reducing the length performs an implicit :ref: `delete` on each of the removed elements. If you try to resize a non-dynamic array that isn't in storage, you receive a `Value must be an lvalue` error.

Solidity Types - Array Members

push:

Dynamic storage arrays and `bytes` (not `string`) have a member function called `push` that you can use to append an element at the end of the array. The element will be zero-initialised. The function returns the new length.

Solidity Types - Array Members

pop:

Dynamic storage arrays and `bytes` (not `string`) have a member function called `pop` that you can use to remove an element from the end of the array. This also implicitly calls :ref:`delete` on the removed element.

Solidity Types - Structs

```
struct Funder {
    address addr;
    uint amount;
}

struct Campaign {
    address payable beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
}
```

Solidity Types - Structs

```
uint numCampaigns;
mapping (uint => Campaign) campaigns;
```

Solidity Types - Structs

```
function newCampaign(address payable beneficiary, uint goal) public returns (uint campaignID) {
    campaignID = numCampaigns++; // campaignID is return variable
    // Creates new struct in memory and copies it to storage.
    // We leave out the mapping type, because it is not valid in memory.
    // If structs are copied (even from storage to storage), mapping types
    // are always omitted, because they cannot be enumerated.
    campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
}
```

Solidity Types - Structs

```
function contribute(uint campaignID) public payable {
    Campaign storage c = campaigns[campaignID];
    // Creates a new temporary memory struct, initialised with the given values
    // and copies it over to storage.
    // Note that you can also use Funder(msg.sender, msg.value) to initialise.
    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
    c.amount += msg.value;
}
```

Solidity Types - Structs

```
function checkGoalReached(uint campaignID) public returns (bool reached) {
    Campaign storage c = campaigns[campaignID];
    if (c.amount < c.fundingGoal)
        return false;
    uint amount = c.amount;
    c.amount = 0;
    c.beneficiary.transfer(amount);
    return true;
}
```

Solidity Types - Mappings

You declare mapping types with the syntax `mapping(_KeyType => _ValueType)`. The `_KeyType` can be any elementary type. This means it can be any of the built-in value types plus `bytes` and `string`. User-defined or complex types like contract types, enums, mappings, structs and any array type apart from `bytes` and `string` are not allowed. `_ValueType` can be any type, including mappings.

Solidity Types - Mappings

You can think of mappings as hash tables, which are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's default value. The similarity ends there, the key data is not stored in a mapping, only its `keccak256` hash is used to look up the value.

Solidity Types - Mappings

You can mark variables of mapping type as `public` and Solidity creates a getter for you. The `_KeyType` becomes a parameter for the getter. If `_ValueType` is a value type or a struct, the getter returns `_ValueType`. If `_ValueType` is an array or a mapping, the getter has one parameter for each `_KeyType`, recursively. For example with a mapping:

Solidity Types - Mappings

```
pragma solidity >=0.4.0 <0.6.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

Solidity Types - delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a mapping is probably a better choice.