Solidity Types: Mapping, Conversion, Value & Reference Types Explained

Reading time	Published	Updated
16 min	Jul 1, 2019	Sep 30, 2019

In Solidity, every state or local variable has a specified type since this language is statically typed. Interactions between them can take place in expressions containing operators.

This tutorial reviews a variety of concepts related to Solidity types. The first section indicates Solidity value types: booleans, integers, fixed point numbers, smart contract addresses, integer literals, etc.

Additionally, we present the function and reference types and dig deeper into the Solidity mapping and its types. The tutorial also reviews the operators involving LValues (a) and indicates the possible conversions with Solidity types.

Our interactive course on Solidity and smart contract creation introduces you to these important concepts as well.

Contents

Solidity Types: Main Tips

- Solidity **value types** include booleans, integers, fixed point numbers, addresses, contract types, fixed-size byte arrays, rational and integer literals, and enums.
- Reference types such as **arrays** and **structs** can be stored in these options: `memory` , `storage` , and `calldata` .
- Mapping in Solidity is seen as **hash tables** (initialized virtually) with the goal to contain each potential key and map it to a value (its byte-representation should consist of zeroes only).

Value Types

Variables of these Solidity value types are always passed by **value**. In other words, such variables are copied when they are used in **assignments** or as function **arguments**.

Booleans

Booleans of the Solidity value types can be either `true` or `false` . The boolean is defined with the `bool` keyword.

It works with these operators:

- `!` (logical negation)
- `&&` (logical conjunction, AND)
- `||` (logical disjunction, OR)
- `==` (equality)
- `!=` (inequality)

Integers

There are two main Solidity types of integers of differing sizes:

- `int` - signed integers.
- `uint` - unsigned integers.

Speaking of size, to specify it, you have keywords such as `uint8` up to `uint256` , that is, of 8 to 256 bits. The simple `uint` and `int` are similar to `uint256` and `int256` , respectively.

Integers work with the following operators:

Comparison operators (evaluates to `bool`)

- `<=` (less than or equal)
- `<` (less than)
- `==` (equal to)
- `!=` (not equal to)
- `>=` (greater than or equal)

- `^` (bitwise XOR (exclusive OR))
- `~` (bitwise NOT)

Arithmetic operators

- `+` (addition)
- `-` (subtraction)
- unary `-` (subtract on a single operand)
- unary `+` (add on a single operand)
- `*` (multiply a single operand)
- `/` (division)
- `%` (remainder (of division))
- `**` (exponentiation)
- `<<` (left shift)
- `>>` (right shift)

Fixed Point Numbers

There are two types of fixed point numbers:

- `fixed` - signed fixed point number.
- `ufixed` - unsigned fixed point number.

This value type also can be declared keywords such as `ufixedMxN` and `fixedMxN`. The `M` represents the amount of bits that the type takes, with `N` representing the number of decimal points that are available. `M` has to be divisible by 8, and a number from 8 to 256. `N` has to be a value between 0 and 80, also being inclusive.

Note: fixed point numbers can be declared in Solidity, but they are not completely supported by this language.

The fixed point numbers function with these operators:

Comparison operators (evaluates to bool)

Arithmetic operators

- `+` (addition)
- `-` (subtraction)
- unary `-` (subtract on a single operand)
- unary `+` (add on a single operand)
- `*` (multiply a single operand)
- `/` (division)
- `%` (remainder (of division))

Addresses

The address Solidity value type has **two** similar kinds:

- `address` holds a 20-byte value (size of an Ethereum address).
- `address payable` is the same as address, but have `transfer` and `send` members.

Note: there are two distinctions between smart contract addresses - the **address payable** can receive Ether, while simple **address** cannot.

Implicit conversions of addresses:

- From `address payable` to `address` : allowed.
- From `address` to `address payable` : not allowed. This type of conversion is only possible with intermediate conversion to `uint160` .
- Address literals can be converted to `address payable` .

Explicit conversions to and from address are permitted for integers, integer literals, contact types and bytes20. However, Solidity prevents the conversions of `address payable(x)` .

The `address(x)` can be converted to `address payable` in cases when `x` is of integer, fixed bytes type, or a literal or a contract that has a `payable` fallback function. When `x` is a contract without the `payable` fallback function, the `address(x)` is of type address.

call the transfer function on **msg.sender** which is an **address payable**.

Members of Address

The `balance` property **queries the balance** of an address, while Ether can be sent to addresses with the `transfer` function.

The `transfer` function queries the balance of an address by applying the property balance and sending Ether (in units of `wei`) to a payable address:

Example

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

When the balance of current contracts **is not sufficient** enough or when the receiver **rejects** the transfer, the `transfer` function fails. It reverts on failure.

Remember: the **send** function is not a safe alternative for **transfer**. The send fails to transfer when the call stack depth reaches 1024 or when the recipient no longer has gas.

Gain more **direct control** over encoding or interface with contracts (not adhere to the ABI) with `call` , `delegatecall` and `staticcall` functions.

They accept one `bytes memory` parameter, deliver the success condition as a boolean and the returned data. For encoding of structured data, use `abi.encode` , `abi.encodePacked` , `abi.encodeWithSelector` and `abi.encodeWithSignature` :

Example

```
bytes memory payload = abi.encodeWithSignature("register(string)",
"MyName");
(bool success, bytes memory returnData) =
address(nameReg).call(payload);
require(success);
```

```
address(nameReg).call.gas(1000000)
(abi.encodeWithSignature("register(string)", "MyName"));
```

You can **manipulate** the Ether value as well:

Example Copy

```
address(nameReg).call.value(1 ether)
(abi.encodeWithSignature("register(string)", "MyName"));
```

It is possible to **combine** these modifiers. Their order is not important:

Example Copy

```
address(nameReg).call.gas(1000000).value(1 ether)
(abi.encodeWithSignature("register(string)", "MyName"));
```

Contract Types

All contracts **define** their type. It is possible to implicitly convert contracts to the contracts they inherit from. Contracts can be changed from and to `address` with **explicit conversion**.

The explicit conversion from and to `address payable` is permitted **only** when the contract type contains the `payable` fallback function.

Note: contracts do not work with operators.

Fixed-size Byte Arrays

The value types `bytes1` , `bytes2` , `bytes3` , ..., `bytes32` contain a **sequence of bytes** (from 1 to 32).

Operators that can be applied to this Solidity value type:

- **Comparisons:** `<=` , `<` , `==` , `!=` , `>=` , `>` (evaluate to bool)
- **Bit operators:** `&` , `|` , `^` (bitwise exclusive or), `~` (bitwise

Note: there is one possible parameter of **.length**. It provides the fixed length of the byte array (read-only).

Dynamically-Sized Byte Array

- `bytes` are dynamically-sized byte array and **is not** one of the Solidity value types.
- `string` is a dynamically-sized UTF-8-encoded string and **is not** a value type.

Rational and Integer Literals

Integer literals are seen as **decimals**. They are created from a sequence of numbers (0-9).

Remember: prior to Solidity 0.4.0 version, division on integer literals **truncated**. Now, such a division converts into a rational number (5 / 2 is not 2, but 2.5).

Enums

Enums generate **user-defined** Solidity types. The explicit conversion is possible **to** and **from** all integer types, but the implicit conversion is not.

Note: during the **explicit conversion** from **integer**, it is confirmed whether the **value at runtime** is inside the range of the enum. If this is not confirmed, a failing assert occurs.

Example Copy

```
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice =
ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

Since enum types are not part of the ABI, the signature of `getChoice` will automatically be modified to `getChoice() returns (uint8)` for all matters external to Solidity.

The integer type used is just large enough to hold all enum values, i.e., if you have more than 256 values, `uint16` will be used and so on.

Function Types

Solidity function types represent types of functions. Function type variables can be assigned from functions. To **designate** functions to and **return** functions from function calls, use function parameters.

Note: functions are **public** by nature unless they are applied as the names of types. Then, the functions are internal.

The **usage** of function members is illustrated in this code example:

Example Copy

```
contract Example {
    function f() public payable returns (bytes4) {
        return this.f.selector;
    }

    function g() public {
        this.f.gas(10).value(800)();
    }
}
```

Members that the **public** and **external** functions can have:

- `.selector` : retrieves the ABI function selector.
- `.gas(uint)` : retrieves a callable function object. After that function is called, it sends the indicated amount of gas to the target function.
- `value(uint)` : retrieves a callable function object. After that function is called, it sends the specified amount of wei to the target function.

The following example shows the usage of **internal function**. It can be applied to internal library functions because they will belong to the same code context:

Example Copy

```
library ArrayUtils {

    function map(uint[] memory self, function (uint) pure returns
(uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }

    function range(uint length) internal pure returns (uint[]
memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}


contract Pyramid {
    using ArrayUtils for *;

    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }

    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

```
contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external
callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted
source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST = Oracle(0x1234567); //
known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
}
```

Call internal functions only on the current contract since they cannot be completed **outside the context** of the current contract.

External functions contain **signatures** of functions and addresses: such functions can be passed and returned from external function calls.

Notation of function types:

Example Copy

Tip: the return types cannot be empty. To set the function to not return anything, **remove** the **<return types>** part of the code.

Function types are **internal** by default. Therefore, it is possible to remove the internal keyword. Remember that this is only applicable for function types.

Reference Types

Solidity reference types have to be handled with **more caution** than value types. It is crucial to clearly **indicate** the data area where the reference type is stored: `memory` , `storage` or `calldata` . Reference type values are manipulated via more than one different name.

Data Location and Assignment Behavior

Every reference type contains information on where it is **stored**. There are three possible options: `memory` , `storage` , and `calldata` . The set location is important for semantics of assignments, not only for the persistence of data:

- Assignments between `storage` and `memory` (or from `calldata`) always generate an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a local storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

Example

Copy

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    uint[] x;

    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to
storage
        uint[] storage y = x; // works, assigns a pointer, data
location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a
new temporary /
        // unnamed array in storage, but storage is "statically"
allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the
pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy
in memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

Arrays

Arrays can have **fixed** or **dynamic** sizes for the compiling process. The array with fixed size `k` and element type `T` is combined as `T[k]`. The dynamically-sized array is `T[]`.

An array consisting of 6 dynamic arrays of `uint` looks like this: `uint[] [6]`. By default in Solidity, x[3] array consists of three elements of type despite the fact that it can be an array.

Note: array elements can have any type. However, there are some limitations: mappings must be stored in the **storage** data location and public functions are to have ABI type parameters.

memory.

Remember: differently than **storage** arrays, size of **memory** arrays cannot be manipulated. Determine the size beforehand or generate a new array and transfer all elements to it.

Example Copy

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

Array Literals

A list of one or multiple expressions, separated by **commas** and placed in **square brackets** ([...]) is an array literal. It is a statically-sized memory array.

The following example shows that the type of `[1, 2, 3]` is `uint8[3] memory`. Since every constant is of `uint8` type, the first element must be converted to `uint` before it can be `uint[3] memory`.

Example Copy

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] memory) public pure {
        // ...
    }
}
```

to `uint[] memory`):

Example

Copy

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    function f() public {
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

Note: there are plans to remove this restriction, but it might cause additional problems.

Array Members

Arrays have these **members**:

- `length` - indicates the number of elements. For memory arrays, the length is fixed after they are generated. However, it can be dynamic and can rely on runtime parameters. The length is set to dynamically-sized arrays to change their size.
- `push` - attaches an element at the end of the dynamic storage arrays and bytes (not string). The newly-added element is zero-initialized.
- `pop` removes an element at the end of the dynamic storage arrays and bytes (not string).

Example

Copy

```
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but
a
    // dynamic array of pairs (i.e. of fixed size arrays of length
two).
    // Because of that, T[] is always a dynamic array of T, even if
T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] m_pairsOfFlags;

    // newPairs is stored in memory - the only possibility
    // for public contract function arguments
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // assignment to a storage array performs a copy of
``newPairs`` and
        // replaces the complete array ``m_pairsOfFlags``.
        m_pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // stores a reference to ``s`` in ``g``
        StructType storage g = s;
        // also changes ``s.moreInfo``.
        g.moreInfo = 2;
        // assigns a copy because ``g.contents``
        // is not a local variable, but a member of
        // a local variable.
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public
{
        // access to a non-existing index will throw an exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // if the new size is smaller, removed array elements will
be cleared
        m_pairsOfFlags.length = newSize;
    }

    function clear() public {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags.length = 0;
    }
```

```
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 0×08;
        delete m_byteData[2];
    }

    function addFlag(bool[2] memory flag) public returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) public pure returns
(bytes memory) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Inline arrays are always statically-sized and if you
only
        // use literals, you have to provide at least one type.
        arrayOfPairs[0] = [uint(1), 2];

        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(uint8(i));
        return b;
    }
}
```

Structs

Solidity allows to define **new types** in the form of structs. This process is shown in the example below:

Example Copy

```
    // Defines a new type with two fields.
    struct Funder {
        address addr;
        uint amount;
    }

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

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

    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),
        // types that are not valid outside of storage (ex.
mappings and array of mappings)
        // are always omitted, because they cannot be enumerated.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    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;
    }

    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;
    }
}
```

Note: a struct **cannot** have a member of the same type as its own.

In all of the functions in the example, a struct type is assigned to a local variable with data location storage. Therefore, it does not copy a struct. Instead, a reference is stored so that assignments to members of the local variable write to the state.

Like in `campaigns[campaignID].amount = 0` , it is possible to directly access the struct members and not assign it to a local variable.

Mapping Types

Syntax of `(_KeyType ⇒_ValueType)` defines the mapping types.

- The `_KeyType` can be any elementary type (plus bytes and string). However, contract types, enums, mappings, structs, and any array type apart are not permitted.
- _ValueType accepts any type, mappings as well.

Note: Solidity mappings can only have a data location of **storage**. Therefore, they are allowed for state variables.

- It is possible to set state variables of mapping Solidity type as public and the language generates a getter. Then, the `_KeyType` is a parameter for the getter.
- In cases when `_ValueType` is a value type or a struct, the getter returns `_ValueType` . When it is an array or mapping, the getter has one parameter for every `_KeyType` , recursively.

Take a look at the example of mapping Solidity:

Example Copy

```
        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));
        }
    }
```

Operators Involving LValues

The `a` is an LValue (for instance, a variable or something that can be assigned to). It has **operators** as shorthands:

- `a += e` is the same as `a= a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=` and `^=` are defined accordingly.
- `a++` and `a--` is the same as `a += 1` / `a -= 1` but the expression still has the previous a value.
- `--a` and `++a` work the same on a but return the value after the change.

delete

In the following example, `delete x` assigns `x` to 0 and **does not** affect data. `delete data` sets data to 0, does not affect `x`:

Example Copy

```
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x;
        delete data;
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero,
but as uint[] is a complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as
assignments to local variables
        // referencing storage objects can only be made from
existing storage objects.
        assert(y.length == 0);
    }
}
```

- The `delete a` sets the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`.
- It is possible to apply it on arrays. It will set a dynamic array of length zero or static array of the same length with all elements assigned their initial value.
- `delete a[x]` will remove an element at the specified array index (won't change other items and length). This results in a gap in an array.
- It sets a struct with all members reset. The a value after `delete a` is the same as delete does not influence mappings. By deleting a struct, you reset all members that are not related to mappings.

Conversions Between Elementary Types

Implicit Conversions

When you use operators on different Solidity types, the compiler aims to **implicitly convert** one of the operands to the type of the other.

Therefore, the operations execute in the type of one of the operands. The implicit conversion takes place without any issues if it is semantically logic, and data is not lost.

Explicit Conversions

When you are confident that a conversion will take place **correctly**, attempt to perform the explicit type conversion.

Warning: be careful since such actions can render **unpredictable results** and **disregard** some security features.

The example below depicts the conversion of a negative `int` to a `uint` :

Example Copy

```
int  y = -3;
uint x = uint(y);
```

At the end of this conversion, the `x` will have the value of `0xfffff..fd` (64 hex characters). It is -3 in the two's complement representation of 256 bits.

In cases when integers are converted into a **smaller type**, higher-order bits are removed. After the second line is compiled, `b` will be `0x5678` :

Example Copy

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
```

The next example shows the opposite situation when an integer is converted to **larger** Solidity types. It will be padded at the higher order end. After the second line, `b` will be `0x00001234` . Comparison between equal and the original integer is the result:

Example Copy

The fixed-size bytes Solidity types act differently during conversions. After the second line, `b` will be `0x12`. If you try to convert them to smaller type, the sequence will be disturbed:

Example Copy

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a);
```

In cases when fixed-size bytes type is converted to a larger type, it is padded on the right. After the second line, `b` will be `0x12340000`:

Example Copy

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a);
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

During truncating or padding, integers and fixed-size byte types act differently. Therefore, **explicit conversions** between them **are not** permitted (unless their sizes are identical). To convert them from different sizes, intermediate conversions make the truncation and padding rules explicit:

Example Copy

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

Conversions Between Literals and Elementary Types

literals to all integer types if they are big enough:

Example Copy

```
uint8 a = 12; // works
uint32 b = 1234; // works
uint16 c = 0×123456; // fails, since it would have to truncate to
0×3456
```

Fixed-Size Byte Arrays

The implicit conversion from decimal number literals to fixed-size byte arrays **is not allowed**. The hexadecimal number **can be converted**, but only in cases when the hex digits number equals the size of the bytes type.

Note: the conversion from decimal and hexadecimal literals with a value of zero to any fixed-size byte type is possible.

Example Copy

```
bytes2 a = 54321; // not allowed
bytes2 b = 0×12; // not allowed
bytes2 c = 0×123; // not allowed
bytes2 d = 0×1234; // works
bytes2 e = 0×0012; // works
bytes4 f = 0; // works
bytes4 g = 0×0; // works
```

The conversion from string literals or hex strings to fixed-size byte arrays is permitted when the **character number** is the same as the size of the bytes type:

Example Copy

```
bytes2 a = hex"1234"; // works
bytes2 b = "xy"; // works
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

Solidity Types: Summary

- **Values** such as booleans, integers, fixed point numbers, address, contract types, fixed-size byte arrays, rational and integer literals, and enums belong to Solidity value types.
- Arrays and structs belong to the **reference types**. Every single one has to indicate the location where it is stored: either `memory` , `storage` or `calldata` .
- Consider mapping as **hash tables**. Mapping types in Solidity are defined with this syntax: `(_KeyType => _ValueType)` .
- The LValue `a` has the `delete` , `delete a` , and operators as shorthands as well.

← Previous Topic   Next Topic →

Related Code Examples

Solidity

Deriving from Solidity Contracts

Solidity

Solidity Event Call

Solidity

Pragma in Solidity Syntax

Solidity

Use of Solidity Functions

Solidity

Resolving Arguments in Solidity Functions

Solidity

Managing Arguments With Solidity Inheritance

LEARN	WEB DEVELOPMENT	SERVER SIDE	TOOLS
Programming Languages	HTML	PHP	Code Editor
Get Certified	CSS	SQL	Color Picker
Sitemap	JavaScript		Devtools
Code Examples			Git

COURSES

COMPANY

Code Theory

Guides & Tutorials

Free Certifications Online

Online Learning Platforms

Learning Paths

Apply for a Scholarship

Hiring

White Paper

Blog

Earn With Us

BitDegree Reviews

FAQ

Press Releases

LEGAL

Policies & Legal Documents

Privacy Policy

Cookie Policy

Money Back Guarantee & Refund Policy

Terms of Service

INSTRUCTORS & PARTNERS

Instructor Login

Become an Instructor

Become a Partner

TAP

Sponsor a Student

FOLLOW US

Learn to earn: BitDegree free online courses give you the best online education with a gamified experience. Gain knowledge and get your dream job: learn to earn.