Summary

Solidity syntax and semantics:

- arrays
- mappings
- enumerated types
- structs
- function types
- memory flow
- control structures
- assert/require
- exceptions, try/catch
- inheritance
- events

Arrays

Fixed size arrays:

T[k] where **T** is a type and **k** an integer —- length **k** array of elements of type **T**

bytes1, bytes2, …., bytes32 = arrays of 1..32 bytes

if **x: bytesN** then can (read only) access **x** using **x[k]** for 0 ≤ **k** < N

Dynamically sized arrays:

T[] — dynamically sized array of elements of type T

bytes (= **byte[]** but more efficiently encoded)

string (UTF-8 encoded strings, **= bytes,** but no index or length access)

String literals: “hello” or ‘hello’ (no trailing 0’s as in C)

Hexadecimal literals: (hex “0001F2A”)

Multiply indexed arrays:

T[3][5] = a length 5 array with elements of type **T[3]**

Access order is the *reverse* of the declaration order:

if **a : T[2][5]** then **a[4] : T[2]** and **a[4][1] : T**

Dynamic arrays **x: T[]** have members:

x.length : int - the length of the array

x.push(y:T) - push y onto x, increasing its length by 1

x.pop : T

Declaring arrays:

Storage arrays

- can be resized using pop() push(), push(value),

Memory arrays

- are declared to have a fixed size (y) using new. e.g.

uint[] memory x = new uint[](y)

- cannot be resized

Enumerated Types

User defined enumerated types:

Example:

enum Day (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)

Day today ;

Day constant payday = Day.Friday

Maximum number of elements of an enum = 256

Structs

```
enum ShareType = { Common, Preferred }

struct Shareholding {
        address shareholder ;
        ShareType class ;
        uint number ;
        bool has_voted_atAGM ;
}

Shareholding bobs_shares ;

bobs_shares.number = 100
```


Structs and enums can be defined at file level, need not be inside contracts

Mappings

Mappings are semantically like *total* functions, they are implemented like hash tables

The domain (key) must be a built-in value domain, **bytes** or **string,** and cannot be a user defined type.

Initially, the value for any input is the default value for the output domain.


```
mapping(address => uint) public number_of_shares
```


Updating entries:


```
number_of_shares[msg.sender] =  100
```

Mappings and types that contain them can only be saved in *storage*, and not in calldata or memory.

Function Types

A variable can be declared to have a function type:

function (parameter-types)
[internal | external]
[pure | view | payable]
[returns (return-types)]

where the visibility and access control keywords have the usual meaning.
(public/private not applicable for function types)

Functions can be passed as arguments of functions.

Example of function types

```
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]);
  }
}
```

passing a function as an argument

Calling the function

(Example from Solidity docs, Section 3.4)

Accessing the current block/transaction

block.blockhash

block.coinbase

block.difficulty

block.number

block.timestamp

tx.gasprice

tx.origin (sender of the original transaction)

Data Location

Recall that the EVM distinguishes between memory types:
storage (long term), **memory** (runtime) and **calldata** (runtime)

There is no explicit pointer type, but variables of complex object type (structs, arrays and mappings):

- are treated as *references* rather than *values*
- multiple variables can refer to the same object (so updating one can change the other)
- must be given an explicit location (typically **memory** or **storage**)
- **calldata** is valid only for parameters of external functions

References and Assignment

Some rules:

The following types of assignments only create references:

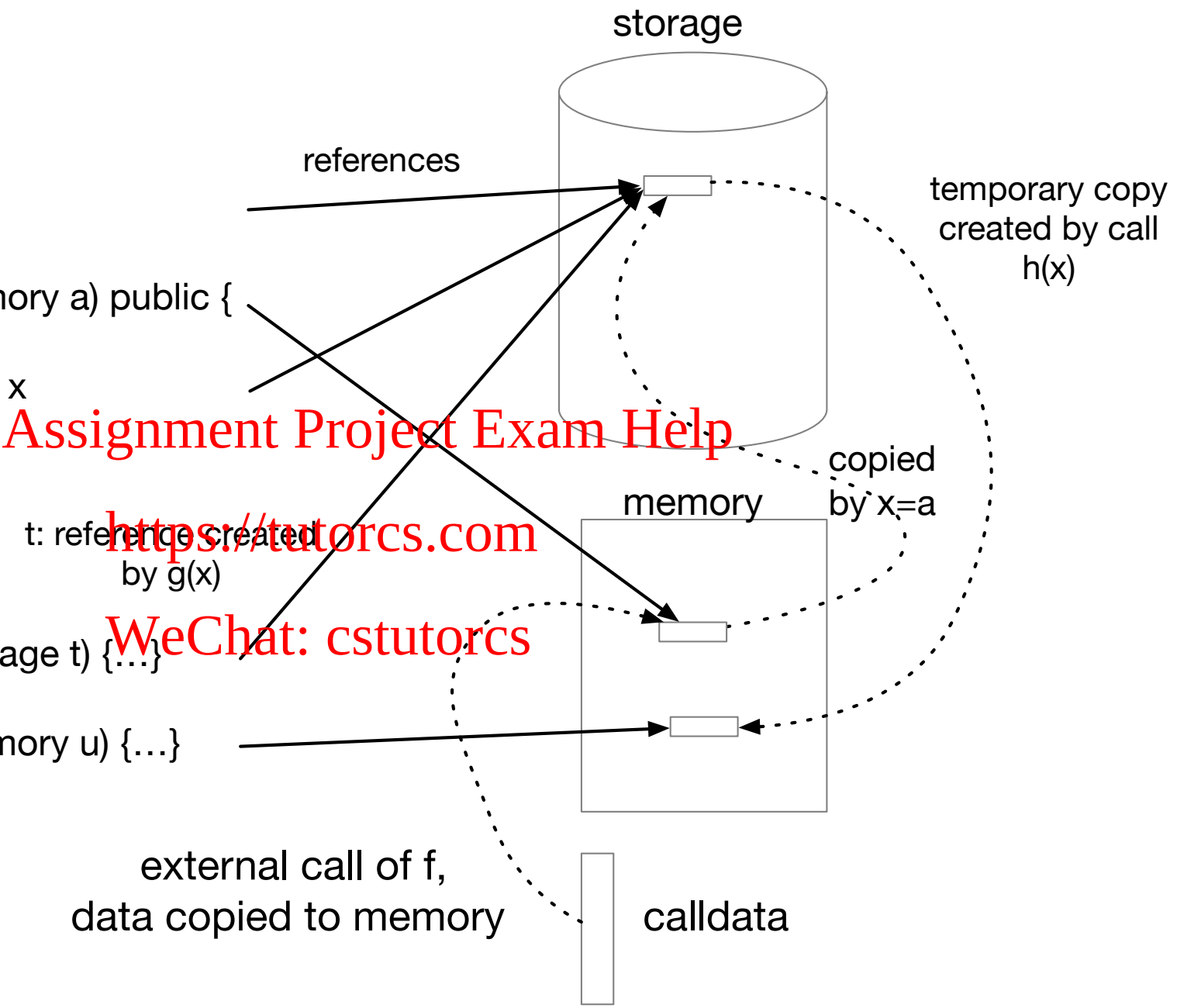
- local storage variable = storage
- memory = memory

The following create independent copies:

- storage = memory or memory = storage
- memory = calldata or storage = calldata
- storage = local variable of a function (on stack)

All other assignments to storage always copy.
(includes assignments to members of storage structs)

```
contract C {

   uint[]  x

   function f(uint[] memory a) public {
      x = a
      uint[] storage y  = x
      g(x);
      h(x)
    }

   function g(uint[] storage t) {…}

   function h(uint[] memory u) {…}

}
```



Control Structures

Solidity borrows its control structures from C and Javascript:

if (<condition>) <statement> [else <statement>]

for (<init statement> ; <condition> ; <post loop part>) <statement>

while (<condition>) <statement>

do <statement> while (<condition>)

<condition> expressions must be of type Bool (no cast from int)

Exceptions

Errors in **external** calls are state-reverting:

— changes made to the state in the contract called are **reverted**

— an exception is flagged to the caller

— if not caught this causes the caller also to revert and abort

NB: exceptions from **internal** function calls always abort the current EVM instance

revert("error message")

aborts the current call by reverting the state changes, returning the error message

returns any unconsumed gas to the caller

Non-aborting EVM call cases:

send, **call**, **delegatecall** and **staticcall** return **false** in case of an exception in the call

NOTE:

send, **call**, **delegatecall** and **staticcall** return **true** if sent to a "non-existent" address*!

This condition needs to be checked before performing these operations

* But what does this mean? See https://github.com/ethereum/solidity/issues/4910

Try/Catch

Exceptions in external calls (but not local computations) can be caught using the try/catch syntax:

```
try <expression with external call>
    returns <variable to capture return value> {
        // code to execute on successful return
        } catch Error(string memory reason) {
            // code to manage catching Error
        } catch Panic(uint panicCode) {
            // code to manage catching Panic
        } catch (bytes memory  lowLevelData) {
            // code to manage exception returning low level data
        }
```

Optional exception types.

catch { //code }
ignores the exception type

Exceptions not caught in the caller “Bubble up”, reverting the caller as well as the function called.

Assert and Require

Operationally, the following both do “abort and revert state changes”, in case the boolean expression evaluates to false, but

require(<boolean-expression>) or **require(<boolean-expression>,”error message”)** translates to a **revert**, and returns unconsumed gas to the caller

assert(<boolean-expression>) eats up all remaining gas balance for the call, and cannot be caught by the caller

It is intended that *program analysis* tools will* handle these differently, treating these statements as generating specification statements from **assert(e)** of the form

“if all the previous **require** statements are true, then **e** must be true here”

You should therefore use **require** to check input validity conditions and state state invariants that all functions are supposed to preserve, and **assert** to check state conditions that should be true on the assumption that the require conditions have been met.

* There is work in progress to build checking for this into the Solidity compiler: **https://medium.com/@leonardoalt/formal-verification-in-solidity-5cbff7b7ff8**

Assert/Require Example

```
pragma solidity >0.4.99 <0.6.0;

contract Sharer {

    function sendHalf(address payable addr) public payable returns (uint balance) {

        require(msg.value % 2 == 0, "Even value required.");

        uint balanceBeforeTransfer = address(this).balance;

        addr.transfer(msg.value / 2);

        // Since transfer throws an exception on failure and cannot call back here,
        // there should be no way for us to still have half of the money.

        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);

        return address(this).balance;
    }
}
// Example from Solidity Docs (section Error Handling ..)
```

Assert-style exceptions

The following errors also cause a revert and consume all remaining gas:

- divide by zero
- array indexed out of bounds
- invalid type conversions
- boolean shift by a negative amount

Modifiers

Modifiers are blocks of code. They can be listed a the function header and applied at the start of the function in order to change its behaviour. (This is a type of *aspect oriented* programming.) Multiple modifiers can be applied (executed left to right)

Typical uses are enforcing access control conditions

Example:


```
address owner ;

modifier onlyOwner {
    require(msg.sender == owner) ;
    _;
}

function changeOwner (address newowner) onlyOwner {
    owner = newowner ;
}
```


_ = “continue with contract code (or the next modifier) from here”

“apply the modifier”

Inheritance

```
abstract contract AbstractContract {

   /* A contract type is abstract when it has some virtual functions without code.
      It just defines an inheritable interface in this case.
      A virtual function can be overriden in subclasses

   */

   function functionName (arguments) public virtual  returns (resultType)

}


contract ContractType2 is AbstractContract, ContractType1, .... {

   /* This inherits non-private variables and functions of AbstractContract, ContractType1, ....
      (re)define an inherited virtual function using override.
      Including virtual allows this definition itself to be overridden.

   */

   function functionName (arguments) public [ virtual ]  override returns (resultType) { // code }

}
```

Events

Events provide a way to write information to the EVM log, to provide the caller with information about what happened during the running of a transaction.

Declaring an event type:

```
event Sent(address from, address to, uint amount)
```


Writing an event to the log:

```
emit Sent(msg.sender, address(this), msg.value)
```

Logs can be written but not read by contracts

Other Aspects

Some other features that we do not cover here in detail:

- Inline EVM assembly
  - this allows optimisations using low level coding at EVM level
  - is as dangerous as it is in other languages (breaks type-safety)

Summary

Solidity syntax and semantics:

- arrays
- mappings
- enumerated types
- structs
- function types
- memory flow
- control structures
- assert/require
- exceptions, try/catch
- inheritance
- events