Secureum Bootcamp Solidity 101 Quiz

October 24, 2021 / patrickd

This is a writeup of the Secureum Bootcamp Solidity 101 Quiz containing solutions and references to the provided study material.
For fairness it was published after submissions to it were closed.

The quiz consisted of 32 questions with a strict timelimit of 16 minutes. The ordering of the questions was randomized, so the numbering here won't match with the numbering elsewhere.

> "User from EOA A calls Contract C1 which makes an external call (CALL opcode) to Contract C2. Which of the following is/are true?"
>
> — 1 of 32

- [x] A. tx.origin in C2 returns A's address
- [ ] B. msg.sender in C2 returns A's address
- [x] C. msg.sender in C1 returns A's address
- [ ] D. msg.value in C2 returns amount of Wei sent from A

▼ Solution

Correct is A, C.

> Block and Transaction Properties:
> msg.sender (address): sender of the message (current call)

> msg.value (uint): number of wei sent with the message
> tx.origin (address): sender of the transaction (full call chain)

from point 73 of Solidity 101 - by Secureum

> The values of all members of msg, including msg.sender and msg.value can change for every external function call. This includes calls to library functions.

from point 74 of Solidity 101 - by Secureum

> “Which of the following is/are true for call/delegatecall/staticcall primitives?”
>
> — 2 of 32

- [x] A. They are used to call contracts
- [ ] B. They only revert without returning success/failure
- [x] C. Delegatecall retains the msg.sender and msg.value of caller contract
- [ ] D. Staticcall reverts if the called contract reads contract state of caller

▼ Solution

Correct is A, C.

> Call/Delegatecall/Staticcall: In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory). With delegatecall, only the code of the given address is used but all other aspects (storage, balance, msg.sender etc.) are taken from the current contract. The purpose of delegatecall is to use library/logic code which is stored in callee contract but operate on the state of the caller contract. With staticcall, the execution will revert if the called function modifies the state in any way

from point 49 of Solidity 101 - by Secureum

> “The gas left in the current transaction can be obtained with”

— 3 of 32

- [ ] A. tx.gas()
- [x] B. gasleft()
- [ ] C. msg.gas()
- [ ] D. block.gaslimit()

▼ Solution

Correct is B.

> Block and Transaction Properties:
> block.gaslimit (uint): current block gaslimit
> tx.gasprice (uint): gas price of the transaction
> gasleft() returns (uint256): remaining gas.

from point 73 of Solidity 101 - by Secureum

> The function gasleft was previously known as msg.gas, which was deprecated in version 0.4.21 and removed in version 0.5.0.

from Block and Transaction Properties of Units and Globally Available Variables - Solidity Docs

> “The default value of”

— 4 of 32

- [x] A. Bool is false
- [x] B. Address is 0
- [x] C. Statically-sized array depends on the underlying type
- [x] D. Enum is its first member

▼ Solution

Correct is A, B, C, D.

> Default Values: A variable which is declared will have an initial default value whose byte-representation is all zeros. The “default values” of variables are the typical “zero-state” of whatever the type is. For example, the default value for a bool is false. The default value for the uint or int types is 0. For statically-sized arrays and bytes1 to bytes32, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, bytes and string, the default value is an empty array or string. For the enum type, the default value is its first member.

from point 39 of Solidity 101 - by Secureum

> “Which of the following is/are true about events?”
>
> — 5 of 32

- [x] A. Events are meant for off-chain applications
- [ ] B. Events can be accessed only by the emitting contract
- [x] C. Indexing event parameters creates searchable topics
- [ ] D. A maximum of three events can have indexed parameters

▼ Solution

Correct is A, C.

> Events: They are an abstraction on top of the EVM’s logging functionality. Emitting events cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible. The Log and its event data is not accessible from within contracts (not even from the contract that created them). Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

from point 27 of Solidity 101 - by Secureum

> Indexed Event Parameters: Adding the attribute indexed for up to three parameters adds them to a special data structure known as “topics” instead of the data part of the log. If you use arrays (including string and bytes) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a

> topic can only hold a single word (32 bytes). All parameters without the indexed attribute are ABI-encoded into the data part of the log. Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

from point 28 of Solidity 101 - by Secureum

> "Function foo() uses block.number. Which of the following is/are always true about foo()?"

— 6 of 32

- [ ] A. It should be marked as pure
- [ ] B. It should be marked as view
- [ ] C. It should be marked as payable
- [x] D. Cannot determine mutability based only on this information

▼ Solution

Correct is D. Since it wouldn't always be true unless you know what other things foo() uses too.

> Function Mutability Specifiers: Functions can be specified as being pure or view:
> view functions can read contract state but cannot modify it. This is enforced at runtime via STATICCALL opcode. The following are considered state modifying: 1) Writing to state variables 2) Emitting events 3) Creating other contracts 4) Using selfdestruct 5) Sending Ether via calls 6) Calling any function not marked view or pure 7) Using low-level calls 8) Using inline assembly that contains certain opcodes.
> pure functions can neither read contract state nor modify it. The following are considered reading from state: 1) Reading from state variables 2) Accessing address(this).balance or address.balance 3) Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data) 4) Calling any function not marked pure 5) Using inline assembly that contains certain opcodes.
> It is not possible to prevent functions from reading the state at the level of the EVM. It is only possible to prevent them from writing to the state via

> STATICCALL. Therefore, only view can be enforced at the EVM level, but not pure.

from point 24 of Solidity 101 - by Secureum

> “Solidity functions”
>
> — 7 of 32

- [ ] A. Can be declared only inside contracts
- [x] B. Can have named return variables
- [x] C. Can have unnamed parameters
- [x] D. Can be recursive

▼ Solution

Correct is B, C, D.

> Free Functions: Functions that are defined outside of contracts are called “free functions” and always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.

from point 26 of Solidity 101 - by Secureum

> Function Return Variables: Function return variables are declared with the same syntax after the returns keyword. The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.

from point 21 of Solidity 101 - by Secureum

> Function parameters: Function parameters are declared the same way as variables, and the name of unused parameters can be omitted. Function parameters can be used as any other local variable and they can also be assigned to.

from point 20 of Solidity 101 - by Secureum

> Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

from point 40 of Solidity 101 - by Secureum

> “Conversions in Solidity have the following behavior”
>
> — 8 of 32

- [ ] A. Implicit conversions are never allowed
- [x] B. Explicit conversions of uint16 to uint8 removes the higher-order bits
- [ ] C. Explicit conversion of uint16 to uint32 adds lower-order padding
- [ ] D. Explicit conversions are checked by compiler for safety

▼ Solution

Correct is B.

> Implicit Conversions: An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. Implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

from point 68 of Solidity 101 - by Secureum

> Explicit Conversions: If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler e.g. int to uint. If an integer is explicitly converted to a smaller type, higher-order bits are cut off. If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end).

from point 69 of Solidity 101 - by Secureum

> “When Contract A attempts to make a delegatecall to Contract B but a prior transaction to Contract B has executed a selfdestruct”
>
> — 9 of 32

- [ ] A. The delegatecall reverts
- [ ] B. The delegatecall returns a failure
- [x] C. The delegatecall returns a success
- [ ] D. This scenario is not practically possible

▼ Solution

Correct is C.

> The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

from point 87 of Solidity 101 - by Secureum

> “If a = 1 then which of the following is/are true?”
>
> — 10 of 32

- [x] A. a += 1 makes the value of a = 2
- [ ] B. b = ++a makes the value of b = 1
- [ ] C. a -= 1 makes the value of a = 1
- [x] D. b = a-- makes the value of b = 1

▼ Solution

Correct is A, D.

> Operators Involving LValues (i.e. a variable or something that can be assigned to)
> a += e is equivalent to a = a + e. The operators -=, *=, /=, %=, |=, &= and ^= are defined accordingly
> a++ and a-- are equivalent to a += 1 / a -= 1 but the expression itself still has the previous value of a

> In contrast, --a and ++a have the same effect on a but return the value after the change

from point 66 of Solidity 101 - by Secureum

> “transfer and send primitives”

— 11 of 32

- [x] A. Are used for Ether transfers
- [x] B. Trigger the receive() or fallback() functions of address
- [ ] C. Always return a value to be checked
- [x] D. Provide only 2300 gas

▼ Solution

Correct is A, B, D.

> The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers via .send() or .transfer(). In the worst case, the receive function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging.

from point 33 of Solidity 101 - by Secureum

> .transfer(uint256 amount) [no return value]: send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable
> .send(uint256 amount) returns (bool): send given amount of Wei to Address, returns false on failure, forwards 2300 gas stipend, not adjustable

from point X of Solidity 101 - by Secureum

> “A contract can receive Ether via”

— 12 of 32

- [x] A. msg.value to payable functions
- [x] B. selfdestruct destination
- [x] C. coinbase transaction
- [x] D. receive() or fallback() functions

▼ Solution

Correct is A, B, C, D.

> A contract without a receive Ether function can receive Ether as a recipient of a coinbase transaction (aka miner block reward) or as a destination of a selfdestruct. A contract cannot react to such Ether transfers and thus also cannot reject them. This means that address(this).balance can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

from point 33.3 of Solidity 101 - by Secureum

> Receive Function: A contract can have at most one receive function, declared using receive() external payable without the function keyword. This is the function that is executed on plain Ether transfers via .send() or .transfer().

from point 33 of Solidity 101 - by Secureum

> Fallback Function: A contract can have at most one fallback function, declared using either fallback () external [payable] or fallback (bytes calldata _input) external [payable] returns (bytes memory _output), both without the function keyword. This function must have external visibility. The fallback function always receives data, but in order to also receive Ether it must be marked payable. In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available.

from point 34 of Solidity 101 - by Secureum

> A Error(string) exception (or an exception without data) is generated in the following situations: If your contract receives Ether via a public function without payable modifier (including the constructor and the fallback function)

from point 91 of Solidity 101 - by Secureum

> “Structs in Solidity”
>
> — 13 of 32

- [x] A. Are user-defined type
- [x] B. Are reference types
- [x] C. Can contain or be contained in arrays and mappings
- [ ] D. None of the above

▼ Solution

Correct is A, B, C.

> Struct Types: They are custom defined types that can group several variables of same/different types together to create a custom data structure. The struct members are accessed using ‘.’ e.g.: struct s {address user; uint256 amount} where s.user and s.amount access the struct members.

from point 30 of Solidity 101 - by Secureum

> Mapping Types: Mappings define key-value pairs and are declared using the syntax mapping(_KeyType => _ValueType) _VariableName. The _KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. _ValueType can be any type, including mappings, arrays and structs.

from point 65 of Solidity 101 - by Secureum

> “The following is/are true about ecrecover primitive”
>
> — 14 of 32

- [x] A. Takes a message hash and ECDSA signature values as inputs
- [ ] B. Recovers and returns the public key of the signature
- [x] C. Is susceptible to malleable signatures
- [ ] D. None of the above

▼ Solution

Correct is A, C. Although internally it first recovers the public key from the signature, it actually returns the address derived from the public key.

> ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address): recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature: r = first 32 bytes of signature, s = second 32 bytes of signature, v = final 1 byte of signature. ecrecover returns an address, and not an address payable.

from point 79.6 of Solidity 101 - by Secureum

> If you use ecrecover, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin has a ECDSA helper library that you can use as a wrapper for ecrecover without this issue.

from point 80 of Solidity 101 - by Secureum

> “Which of the following is/are valid control structure(s) in Solidity (excluding YUL)?”
>
> — 15 of 32

- [x] A. if
- [x] B. else
- [ ] C. elif
- [ ] D. switch

▼ Solution

Correct is A, B. "elif" is specific to Python and switch has not been implemented in Solidity yet

> Control Structures: Solidity has if, else, while, do, for, break, continue, return, with the usual semantics known from C or JavaScript

from point 85 of Solidity 101 - by Secureum

> “Address types”
>
> — 16 of 32

- [ ] A. Can always receive Ether
- [x] B. Have members for balance, call, code
- [x] C. Can be converted to uint160 or contract types
- [ ] D. Can be added and subtracted

▼ Solution

Correct is B, C.

> Address Type: The address type comes in two types: (1) address: Holds a 20 byte value (size of an Ethereum address) (2) address payable: Same as address, but with the additional members transfer and send. address payable is an address you can send Ether to, while a plain address cannot be sent Ether.

from point 45 of Solidity 101 - by Secureum

> Members of Address Type:
> address.balance (uint256): balance of the Address in Wei
> address.code (bytes memory): code at the Address (can be empty)
> address.call(bytes memory) returns (bool, bytes memory): issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

from point 46 of Solidity 101 - by Secureum

> Conversions: Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(address). Explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types.

from point 45.2 of Solidity 101 - by Secureum

> TypeError: Operator - not compatible with types address and int_const 1. Arithmetic operations on addresses are not supported. Convert to integer first before using them.

from playing around with remix Remix

> “If the previous block number was 1000 on Ethereum mainnet, which of the following is/are true?”
>
> — 17 of 32

- [x] A. block.number is 1001
- [x] B. blochhash(1) returns 0
- [x] C. block.chainID returns 1
- [ ] D. block.timestamp returns the number of seconds since last block

▼ Solution

Correct is A, B, C. Block number is the number of the block that is currently being mined, the next one. Block number 1 was too long ago and its hash can no longer be accessed due to scaling reasons. Mainnet ID Chain is 1.

> Block and Transaction Properties:
> blockhash(uint blockNumber) returns (bytes32): hash of the given block - only works for 256 most recent, excluding current, blocks
> block.chainid (uint): current chain id
> block.number (uint): current block number
> block.timestamp (uint): current block timestamp as seconds since unix epoch

from point 73 of Solidity 101 - by Secureum

> “If we have an array then its data location can be”
>
> — 18 of 32

- [x] A. memory and its persistence/scope will be the function of declaration
- [x] B. storage and its persistence/scope will be the entire contract
- [x] C. calldata and it will only be readable
- [ ] D. None of the above

▼ Solution

Correct is A, B, C.

> Reference Types & Data Location: Every reference type has an additional annotation — the data location where it is stored. There are three data locations: memory, storage and calldata.
> memory: whose lifetime is limited to an external function call
> storage: whose lifetime is limited to the lifetime of a contract and the location where the state variables are stored
> calldata: which is a non-modifiable, non-persistent area where function arguments are stored and behaves mostly like memory. It is required for parameters of external functions but can also be used for other variables.

from point 55 of Solidity 101 - by Secureum

> "delete varName; has which of the following effects?"
>
> — 19 of 32

- [x] A. varName becomes 0 if varName is an integer
- [ ] B. varName becomes true if varName is a boolean
- [x] C. No effect if varName is a mapping
- [ ] D. Resets all struct members to their default values irrespective of their types

▼ Solution

Correct is A, C.

> delete a assigns the initial value for the type to a
> For integers it is equivalent to a = 0
> For structs, it assigns a struct with all members reset
> delete has no effect on mappings. So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

from point 67 of Solidity 101 - by Secureum

> Default Values: A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the

> typical “zero-state” of whatever the type is. For example, the default value for a bool is false.

from point 39 of Solidity 101 - by Secureum

> “Which of the following is/are valid function specifier(s)?”
>
> — 20 of 32

- [x] A. internal
- [x] B. pure
- [x] C. payable
- [ ] D. immutable

▼ Solution

Correct is A, B, C.

> Function Visibility Specifiers: Functions have to be specified as being public, external, internal or private

from point 23 of Solidity 101 - by Secureum

> Function Mutability Specifiers: Functions can be specified as being pure or view

from point 24 of Solidity 101 - by Secureum

> If your contract receives Ether via a public function without payable modifier (including the constructor and the fallback function)

from point 91.3 of Solidity 101 - by Secureum

> State **Variables** can be declared as constant or immutable.

from point 17.1 of Solidity 101 - by Secureum

> “Function visibility”

— 21 of 32

- [x] A. Goes from private-internal-external-public in decreasing restrictive order (i.e. private being the most restrictive)
- [ ] B. Goes from internal-private-external-public in decreasing restrictive order (i.e. internal being the most restrictive)
- [ ] C. May be omitted to default to internal in the latest 0.8.0+ compiler versions
- [ ] D. None of the above

▼ Solution

Correct is A. Default visibility was public but is required in current Solidity versions.

> Function Visibility Specifiers: Functions have to be specified as being public, external, internal or private:
> public: Public functions are part of the contract interface and can be either called internally or via messages.
> external: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function f cannot be called internally (i.e. f() does not work, but this.f() works).
> internal: Internal functions can only be accessed internally from within the current contract or contracts deriving from it
> private: Private functions can only be accessed from the contract they are defined in and not even in derived contracts

from point 23 of Solidity 101 - by Secureum

> “For error handling”

— 22 of 32

- [x] A. require() is meant to be used for input validation
- [ ] B. require() has a mandatory error message string
- [x] C. assert() is meant to be used to check invariants

- [x] D. revert() will abort and revert state changes

▼ Solution

Correct is A, C, D.

> assert(bool condition): causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.
> require(bool condition): reverts if the condition is not met - to be used for errors in inputs or external components.
> require(bool condition, string memory message): reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.
> revert(): abort execution and revert state changes
> revert(string memory reason): abort execution and revert state changes, providing an explanatory string

from point 78 of Solidity 101 - by Secureum

> The assert function creates an error of type Panic(uint256). Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input.

from point 88 of Solidity 101 - by Secureum

> “Which of the following is/are true?”
>
> — 23 of 32

- [ ] A. Constant state variables can be initialized within a constructor
- [ ] B. Immutable state variables are allocated a storage slot
- [x] C. Gas costs for constant and immutable variables is lower
- [x] D. Only value types can be immutable

▼ Solution

Correct is C, D.

> For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared.

from point 17.2 of Solidity 101 - by Secureum

> The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

from point 17.4 of Solidity 101 - by Secureum

> Compared to regular state variables, the gas costs of constant and immutable variables are much lower

from point 18 of Solidity 101 - by Secureum

> “Integer overflows/underflows in Solidity”
>
> — 24 of 32

- [ ] A. Are never possible because of the language design
- [x] B. Are possible but prevented by compiler added checks (version dependent)
- [x] C. Are possible but prevented by correctly using certain safe math libraries
- [ ] D. Are possible without any mitigation whatsoever

▼ Solution

Correct is B, C.

> Integers in Solidity are restricted to a certain range. For example, with uint32, this is 0 up to 2**32 - 1. There are two modes in which arithmetic is performed on these types: The “wrapping” or “unchecked” mode and the “checked” mode. By default, arithmetic is always “checked”, which means that if the result of an operation falls outside the value range of the type, the call is reverted through a failing assertion. You can switch to “unchecked” mode using unchecked . This was introduced in compiler version 0.8.0.

from point 43 of Solidity 101 - by Secureum

> “Which of the following is true about mapping types in mapping(_KeyType => _ValueType)?”
>
> — 25 of 32

- [ ] A. _KeyType can be any value or reference type
- [x] B. _ValueType can be any value or reference type
- [x] C. Can only have storage (not memory) as data location
- [ ] D. Can be iterated over natively (i.e. without implementing another data structure)

▼ Solution

Correct is B, C.

> The _KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. _ValueType can be any type, including mappings, arrays and structs. They can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that.

from point 65 of Solidity 101 - by Secureum

> “receive() and fallback() functions”
>
> — 26 of 32

- [x] A. Can rely only on 2300 gas in the worst case
- [x] B. May receive Ether with payable mutability
- [ ] C. Are mandatory for all contracts
- [x] D. Must have external visibility

▼ Solution

Correct is A, B, D.

> Receive Function: A contract can have at most one receive function, declared using receive() external payable without the function keyword. This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability. In the worst case, the receive function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging. A contract without a receive Ether function can receive Ether as a recipient of a coinbase transaction (aka miner block reward) or as a destination of a selfdestruct.

from point 33 of Solidity 101 - by Secureum

> Fallback Function: A contract can have at most one fallback function, declared using either fallback () external [payable] or fallback (bytes calldata _input) external [payable] returns (bytes memory _output), both without the function keyword. This function must have external visibility. The fallback function always receives data, but in order to also receive Ether it must be marked payable. In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available.

from point 34 of Solidity 101 - by Secureum

> “In Solidity, selfdestruct(address)”
>
> — 27 of 32

- [ ] A. Destroys the contract whose address is given as argument
- [x] B. Destroys the contract executing the selfdestruct
- [ ] C. Sends address's balance to the calling contract
- [x] D. Sends executing contract's balance to the address

▼ Solution

Correct is B, D.

> selfdestruct(address payable recipient): Destroy the current contract, sending its funds to the given Address and end execution.

from point 81.2 of Solidity 101 - by Secureum

> “Which of the following is/are correct?”
>
> — 28 of 32

- [x] A. Solidity file with pragma solidity ^0.6.5; can be compiled with compiler version 0.6.6
- [x] B. Solidity file with pragma solidity 0.6.5; can be compiled with compiler version 0.6.5
- [ ] C. Solidity file with pragma solidity ^0.6.5; can be compiled with compiler version 0.7.0
- [ ] D. Solidity file with pragma solidity >0.6.5 <0.7.0; can be compiled with compiler version 0.7.0

▼ Solution

Correct is A, B.

> Version Pragma: This indicates the specific Solidity compiler version to be used for that source file and is used as follows: “pragma solidity x.y.z;” where x.y.z indicates the version of the compiler.
> Using the version pragma does not change the version of the compiler. It also does not enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.
> A ‘^’ symbol prefixed to x.y.z in the pragma indicates that the source file may be compiled only from versions starting with x.y.z until x.(y+1).z. For e.g., “pragma solidity ^0.8.3;” indicates that source file may be compiled with compiler version starting from 0.8.3 until any 0.8.z but not 0.9.z. This is known as a “floating pragma.”
> Complex pragmas are also possible using ‘>’,’>=‘,’<‘ and ‘<=‘ symbols to combine multiple versions e.g. “pragma solidity >=0.8.0 <0.8.3;”

from point 7 of Solidity 101 - by Secureum

> “The impact of data location of reference types on assignments is”
>
> — 29 of 32

- [ ] A. storage assigned to storage (local variable) makes a copy
- [ ] B. memory assigned to memory makes a copy
- [ ] C. memory assigned to storage creates a reference
- [x] D. None of the above

▼ Solution

Correct is D. They all do the opposite.

> Data Location & Assignment: Data locations are not only relevant for persistence of data, but also for the semantics of assignments.
> Assignments between storage and memory (or from calldata) always create 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.

from point 56 of Solidity 101 - by Secureum

> “Which of the below are value types?”
>
> — 30 of 32

- [x] A. Address
- [x] B. Enum
- [ ] C. Struct
- [x] D. Contract

▼ Solution

Correct is A, B, D.

> Value Types: Types that are passed by value, i.e. they are always copied when they are used as function arguments or in assignments — Booleans, Integers,

> Fixed Point Numbers, Address, Contract, Fixed-size Byte Arrays (bytes1, bytes2, …, bytes32), Literals (Address, Rational, Integer, String, Unicode, Hexadecimal), Enums, Functions.

from point 37 of Solidity 101 - by Secureum

> Reference Types: Types that can be modified through multiple different names. Arrays (including Dynamically-sized bytes array bytes and string), Structs, Mappings.

from point 38 of Solidity 101 - by Secureum

> “Arrays in Solidity”
>
> — 31 of 32

- [x] A. Can be fixed size or dynamic
- [x] B. Are zero indexed
- [x] C. Have push, pop and length members
- [ ] D. None of the above

▼ Solution

Correct is A, B, C.

> Arrays: Arrays can have a compile-time fixed size, or they can have a dynamic size. Indices are zero-based.

from point 57 of Solidity 101 - by Secureum

> Array members:
> length: returns number of elements in array
> push(): appends a zero-initialised element at the end of the array and returns a reference to the element
> push(x): appends a given element at the end of the array and returns nothing
> pop: removes an element from the end of the array and implicitly calls delete on the removed element

from point 58 of Solidity 101 - by Secureum

> “Solidity language is”
>
> — 32 of 32

- [x] A. Statically typed
- [x] B. Object-oriented
- [x] C. Supports inheritance
- [x] D. Supports inline assembly

▼ Solution

Correct is A, B, C, D. Inline assembly support is passively mentioned several times in Solidity 101.

> Solidity is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language.

from point 3 of Solidity 101 - by Secureum

> The syntax and OOP concepts are from C++.

from point 2 of Solidity 101 - by Secureum

In Blockchain Tags Ethereum, Secureum Bootcamp

← Secureum Bootcamp Solidity 201 Q… Secureum Bootcamp Ethereum 101 Qu…