DRAFT: Clarity contract storage and STX transfer

Part 3

Phillip G. Bradford*

November 28, 2024

Abstract

Principal addresses are unique places on the stacks blockchains. *Clarity* smart contracts are addressed by principal addresses. One principal address may have many contracts. Principals have STX balances and Clarity has primitives to transfer STX between principals. Also, Clarity contracts have their own variables and other storage options.

1 Principals

The Clarity memory model has several parts. These parts include memory in a contract and memory on the Stacks chain. The stacks chain contains all Clarity contracts.

A principal is an address on the Stacks blockchain. There are two types of principals:

1. a *standard principal* is an address on the Stacks blockchain.
2. a *contract principal* is the address of a smart contract. Contract principals are standard principals 'dot' contract name.

   For example, `'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.my-contract` is a contract principal. The contract is `my-contract`.

*phillip.bradford@uconn.edu, phillip.g.bradford@gmail.com, University of Connecticut, School of Computing, Stamford, CT USA

These addresses are unique ids. If a principal address starts with "ST" then it is a testing principal.

A principal has an STX-balance. Similarly, each contract of a principal has its own STX-balance.

Listing 1 shows the first three principal addresses and their starting STX-balances. Each of these testing principal addresses starts with 100,000,000,000,000 uSTX. Where 1 million micro-STX, written uSTX is 1 STX.

Listing 1: Starting a local test instance in clarinet

```
Computer> clarinet console
clarity-repl v1.5.4
Enter "::help" for usage hints.
Connected to a transient in-memory database.
+-----------------------------------------+------------------+
| Address                                 | uSTX             |
+-----------------------------------------+------------------+
| ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM | 100000000000000 |
+-----------------------------------------+------------------+
| ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5 | 100000000000000 |
+-----------------------------------------+------------------+
| ST2CY5V39NHDPWSXMW9QDT3HC3GD6Q6XX4CFRK9AG | 100000000000000 |
+-----------------------------------------+------------------+
```

This is a generous testing allotment. The clarinet console shows this amounts to u100000000 STX:

```
>> (/ u100000000000000 u1000000)
u100000000
```

That is, 100,000,000 or 100-million STX.

At the clarinet console, get the value of `tx-sender`. To see the value of the global variable `tx-sender`, enter it in the clarinet console.

```
>> tx-sender
ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM
```

So, let's transfer 500 uSTX from *tx-sender*

```
>> (stx-get-balance tx-sender)
u100000000000000
(stx-get-balance 'ST-xxx.my-contract)
```

Sending funds to a principal,

```
>> (stx-transfer? u500 tx-sender
   'ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5)
Events emitted
"type" :"stx_transfer_event","stx_transfer_event":  .  .  .
(ok true)
```

The (ok true) indicates a successful STX transfer.

Validating this transfer,

```
>> (stx-get-balance tx-sender)
u99999999999500
>> (stx-get-balance 'ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5)
u100000000000500
```

2 Data variables

Variables in a Clarity smart contract are managed with the contract methods,

```
(define-data-var message (string-ascii 15) "Clarity␣is␣a␣
   virtue")

(define-public (get-message) (ok (var-get message)))

>> (contract-call? .c1 get-my-message)
(ok "Clarity is a virtue")
>> (contract-call? .c1 set-my-message "validity␣is␣critical")
(ok true)
>> (contract-call? .c1 get-my-message)
(ok "validity is critical")
```

Variables in a Clarity smart contract have both visibility and scope. For instance,

1. Visibility - what variables in a smart contract are global or local variables, `define-data-var` and in the case of functions `define-public` or `define-private`
2. Scope - local or global scope Where local scope is defined using `let`

   The next example shows visibility and scope together,

   ```
   ;; public functions
   ;;
   ```

```
(define-public (example-local-scope)
  (let ((local-var u5))
    (ok (square (square local-var)))
  )
)
;; private functions
;;
(define-private (square (x uint))
    (* x x))
```

3 Lists

Lists are very usefull data structures. Lisp languages are list-based.

Clarity allows lists of lists.

1. All lists have fixed length
2. All elements of a list are the same type

4 Maps

Maps are associative data structures holding key-value pairs. So, `define-map` takes three arguments. The first argument is the name of the map. The `key-type` argument holds the keys. The `value-type` holds the values.

```
>> (define-map map-name key-type value-type)
```

This highlights how Clarity creates maps between types. The next map maps principals to unsigned integer balances. These balances cannot be negative.

```
;; data maps
;;
;;; A map that creates a principal => uint relation.
(define-map balances principal uint)
```

```
;;
;;
;; Set the "balance" of the tx-sender to u500. - in a
   private/public function
(map-set balances tx-sender u500)

;; Retrieve the balance - in a private/public/read-only
   function
(map-get? balances tx-sender))
```

The text-book example of increasing a contract value and then getting its new value is next.

Listing 2: A map counting a principals holdings

```
;; data maps
;;
(define-map counters principal uint)

;; public functions
;;
(define-public (count-up)
    (begin
        (map-set balances tx-sender (+ (get-count tx-sender)
           u1))
        (ok true)
    )
)

;; read only functions
;;
(define-read-only (get-count (who principal))
    (default-to u0 (map-get? balances who)))
```

Listing 3: Example 2

```
>> (contract-call? .c1 count-up)
>> (contract-call? .c1 count-up)
>> (contract-call? .c1 get-count tx-sender)
```

Maps are very useful in many contexts. The Clarity `print` function is only defined in contracts.

```
(define-map mymap int int)
```

```
(define-public (map-add-pair (x int) (y int))
        (ok (map-set mymap x y)))

(define-public (print-map (x int))
        (begin
                (print (map-get? mymap x))
                (ok true)))

(define-map squares { x: int } { square: int })
(define-private (add-entry (x int))
  (map-insert squares { x: 2 } { square: (* x x) }))
```

A principal address that starts with "ST" is a test-contract.

Next change `tx-sender`,

```
>> ::set_tx_sender STNHKEPYEPJ8ET55ZZ0M5A34J0R3N5FM2CMMMAZ6
>>  (contract-call?
   'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.c1 count-up)
>>  (contract-call?
   'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.c1 count-up)
>>  (contract-call?
   'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.c1 get-count
   tx-sender)
```

What is happening?

5 Exercises

The goal is to complete the next contract.

Listing 4: Basic contract

```
(define-constant err-invalid-caller (err u1))

(define-map authorised-callers principal bool)
(define-map recipients principal bool)

(map-set recipients tx-sender true)
(map-set authorised-callers
   'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2 true)

(define-private (is-valid-caller (caller principal))
    ;; Implement.
        ;;
        ;;
)

(define-public (delete-recipient (recipient principal))
    (if (is-valid-caller tx-sender)
        (ok (map-delete recipients recipient))
        err-invalid-caller
    )
)
```

It may also be useful to check authorized-callers from outside the contract owner. Not a best-practice, but fun to try.

References

[1] Harold Abelson, Gerald Jay Sussman, with Julie Sussman: *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996.

[2] Marvin Janssen with contributions of lnow, Mike Cohen and Albert Catama: *Clarity of Mind*, https://book.clarity-lang.org/ 2023-04-11.

[3] Daniel P. Friedman and Matthias Felleisen: *The Little Schemer*, Fourth edition, MIT Press, 1996.

[4] Kenny Rogers: *Building an NFT with Stacks and Clarity*, https://blog.developerdao.com/building-an-nft-with-stacks-and-clarity, 2022-09-01.

[5] Kenny Roger, Joe Bender: Stacks developer workshop: *Web3 for Bitcoin: The What, Why, and How of Building on Stacks*. Web3 for Bitcoin. Wed, Jun 29, 2022.