WEB3CLUBS FOUNDATION LIMITED

Course Instructor: DR. Cyprian Omukhwaya Sakwa
PHONE: +254723584205 Email: cypriansakwa@gmail.com

Foundational Mathematics for Web3 Builders

Implemented in noir

Lecture 112

July 10, 2025

Understanding Key-Value Mappings in Noir

- What are key-value mappings?
- How to store and retrieve data efficiently in Noir.
- Hands-on examples of creating and manipulating mappings in Noir.
- Practical use cases of key-value mappings in Noir.

What is a Key-Value Mapping?

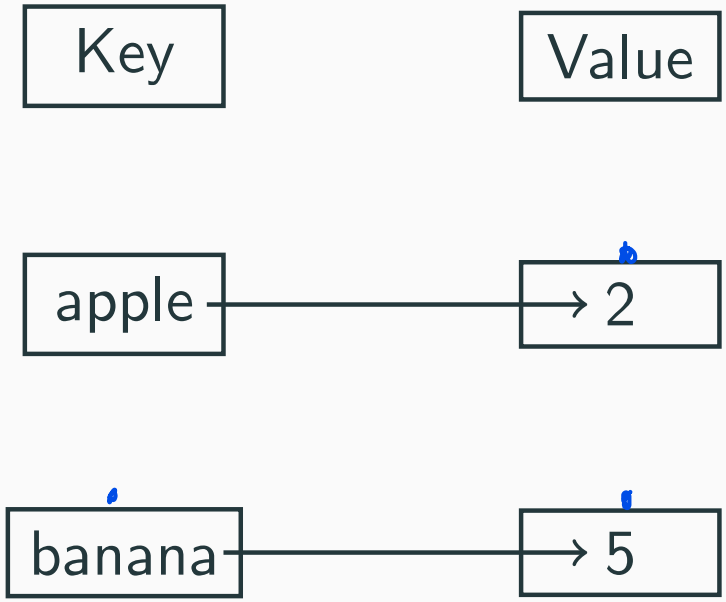
Definition

- A **key-value mapping** is a technique to associate specific **keys** with corresponding **values**.
- The **key** acts as a unique identifier, and the **value** is the associated data.

Why Use Key-Value Mappings?

- **Efficient Lookups**: Quickly retrieve data using keys.
- **Organized Data**: Simplifies the association of related data.
- **Practical Applications**: Useful for word counts, configuration storage, and lookup tables.

Visualizing a Key-Value Mapping



Each key maps to a specific value. In Noir, we simulate Key-Value Maps using fixed-size parallel arrays.

Example:

- `let keys = ["apple   ", "banana  "];`
- `let values = [2, 5];`

Diagram Breakdown

Top Row (Labels):

- `Key` — Unique identifier as `str<8>`
- `Value` — Associated data (e.g., u8)

"apple" → 2

- `let keys = ["apple   ", "banana  "];`
- `get_value(keys, values, "apple   ") = 2`

"banana" → 5

- `get_value(keys, values, "banana  ") = 5`

Creating a Mapping in Noir

Step 1: Define Keys and Values

```
let keys = ["apple␣␣␣", "banana␣
    ␣"]; // Fixed-length strings
let values = [2, 5];
                    //
        Corresponding integer values
```

Explanation:

- `keys`: An array of fixed-length strings (`str<N>`) to represent the keys.
- `values`: An array of integers (`[u8; N]`) to represent the associated values.

Adding Key-Value Pairs

Step 2: Initialize Data

```
let keys = ["apple␣␣␣", "banana␣␣"]; //
    Fixed-length strings
let values = [2, 5];                  //
    Corresponding integer values
```

Explanation:

- Data is initialized as two parallel arrays: one for keys and one for values.
- Each key corresponds to a value at the same index in the arrays.

```
fn main(key: pub str<8>) -> pub u8 {
        // Step 1: Static key-value arrays
        let keys = ["apple␣␣␣", "banana␣␣"]; //
            Each key must be exactly 8 characters
        let values = [2, 5];

        // Step 2: Lookup the value associated
            with the dynamic key
        let result = get_value(keys, values, key
            );

        // Step 3: Return result (no longer a
            constant)
        result
}
```

```
fn get_value(keys: [str<8>; 2], values: [u8; 2],
    key: str<8>) -> u8 {
        let mut result = 0;
        for i in 0..2 {
            if keys[i] == key {
                result = values[i];
            }
        }
        result
}
```

Test Your Understanding

Question: What value would we get if we queried the arrays with the key `"banana "`?

Answer: `5`

Why? The key `"banana "` appears in the array at index 1 and maps to the value 5.

Querying the Key `"banana"`

Step 1: Edit `Prover.toml`

Ensure the key is exactly 8 characters (padded with spaces if needed):

Prover.toml

```
[public]
key = "banana "
```

Step 2: Run the Circuit from Terminal

Navigate to the example folder and execute the circuit:

Terminal Command

```
nargo execute
```

Expected Output:

Result

```
Circuit output:  Vec([Field(5)])
```

This shows that the key `"banana"` maps to the value `5`.

Automating Key Input and Execution

Shell Script: Automatically insert a padded key and run `nargo execute`

```
#!/bin/bash

# Ask for key input
read -p "Enter key to query (max 8 
    characters): " raw_key

# Pad to exactly 8 characters
padded_key=$(printf "%-8s" "$raw_key")

# Write to Prover.toml
cat <<EOF > Prover.toml
key = "$padded_key"
```

```
EOF

# Execute the circuit
echo "Running␣nargo␣execute␣with␣key:␣'
    $padded_key'"
nargo execute
```

Usage:

- Save as `run_query.sh` in your example folder
- Make executable: `chmod +x run_query.sh`
- Run with: `./run_query.sh`

When to Use Key-Value Mapping

Advantages

- **Efficiency**: O(1) average time complexity for lookups and insertions.
- **Organization**: Simplifies data management by associating values with unique keys.
- **Versatility**: Can be used for various purposes like counting occurrences, storing configurations, or creating lookup tables.

Use Cases

- Counting word occurrences in a dataset.
- Storing user preferences or settings.
- Implementing caching mechanisms for faster data access.

Retrieving Values

Step 3: Access Data by Key

```
fn get_value(keys: [str<8>; 2], values: [u8; 2],
    key: str<8>) -> u8 {
        let mut result = 0;
        for i in 0..2 {
                if keys[i] == key {
                        result = values[i];
                }
        }
        result
}

                let apple_count = get_value(keys
                    , values, "apple␣␣␣");
```

Explanation:

Modifying Values

Step 4: Update an Existing Key

```
let mut values = [2, 5];
values[0] = 10; // Update the
    value for "apple"
```

Explanation:

- Values can be directly updated by index.
- For example, updating the value at index 0 changes the value for "apple".

Removing Entries

Step 5: Simulate Key Removal

```
let mut values = [2, 5];
values[1] = 0; // Simulate removing "banana"
```

Explanation:

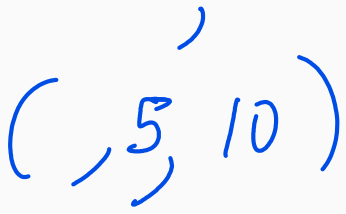
- Removing a key-value pair can be simulated by setting the value to a default (e.g., 0).
- Noir does not support dynamic data structures, so the size of the array remains fixed.

Counting Occurrences Example

Example: Word Frequency Counter

```
fn main(words: pub [str<8>; 6]) -> pub [u8; 3] {
        let mut counts = [0, 0, 0]; //
            Initialize counts to 0

        for word in words {
                if word == "apple␣␣␣" {
                        counts[0] += 1;
                } else if word == "banana␣␣" {
                        counts[1] += 1;
                } else if word == "cherry␣␣" {
                        counts[2] += 1;
                }
```



```
        }

        counts
}
```

Explanation:

- Counts are stored in a fixed-size array corresponding to predefined keys.
- Each word is compared to the predefined keys, and the corresponding count is incremented.

Output Example:

```
Word Frequencies: [3, 2, 1]
```

Real-World Use Case: Configuration Storage

Example: Storing Application Settings

```
fn main(key: pub str<8>) -> pub str<8> {
        // Define keys and values as fixed-size
           arrays
        let keys = ["theme␣␣␣", "language"];
        let values = ["dark␣␣␣␣", "en␣␣␣␣␣␣"];

        // Retrieve the value associated with
           the user-provided key
        let result = get_value(keys, values, key
           );

        result
```

```
}

fn get_value(keys: [str<8>; 2], values: [str<8>;
    2], key: str<8>) -> str<8> {
        let mut result = "␣␣␣␣␣␣␣␣"; // Default
            empty string (8 chars)
        for i in 0..2 {
                if keys[i] == key {
                        result = values[i];
                }
        }
        result
}
```

This Noir code defines a simple key-value lookup circuit — like a very basic HashMap — where the user provides a key as input, and the circuit returns the associated value.

Explanation:

- This circuit simulates a simple key-value lookup — like a mini HashMap.
- It receives a public input key and returns the associated value.
- All keys and values are fixed-length strings (‘str¡8¿‘).

Main Function

```
fn main(key: pub str<8>) -> pub str<8> {
        let keys = ["theme␣␣␣", "language"];
        let values = ["dark␣␣␣␣", "en␣␣␣␣␣␣␣"];
        let result = get_value(keys, values, key
            );
        result
                }
```

- Accepts a public 8-character key.
- Returns the matching 8-character value.

Helper Function: get_value

```
fn get_value(keys: [str<8>; 2], values: [str<8>;
    2], key: str<8>) -> str<8> {
        let mut result = "␣␣␣␣␣␣␣␣";
        for i in 0..2 {
            if keys[i] == key {
                result = values[i];
            }
        }
        result
}
```

- Iterates through the ‘keys‘ array.
- Compares each with the user-provided ‘key‘.
- Returns the matching value, or an empty string if not found.

Inputs and Outputs

- **Input:** A public str<8> key (e.g., `"theme "`).
- **Output:** A str<8> value (e.g., `"dark "`).

Fixed Mappings:

- `"theme"` $\rightarrow$ `"dark"`
- `"language"` $\rightarrow$ `"en"`

Thus, Key-Value Lookups in Noir

- Uses fixed-length string arrays to simulate key-value storage.
- Public input controls lookup dynamically at proof time.
- Clean, simple demonstration of conditional branching and iteration in Noir.

Next: Extend this to private values, longer maps, or dynamic value handling.

Summary

Key Points Recap

1. Noir does not support `HashMap`, but mappings can be implemented using fixed-size arrays.
2. Use custom functions to retrieve values from keys.
3. Use parallel arrays to simulate key-value storage.
4. Practical applications include **word counting**, **configuration storage**, and **lookup tables**.

Hands-On Practice

Try It Yourself!

1. Create parallel arrays to store student names and their grades.
2. Write code to:
    - Add new students and grades.
    - Retrieve grades by student name.
    - Update a grade for an existing student.
    - Remove a student by setting the grade to `0`.

Challenge:

- Count the frequency of letters in a given string using fixed-size arrays.