WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in noir

Lecture 110

July 3, 2025

Arrays in Noir

What is an Array?

An array is a way to store a fixed number of values of the same type together in a single variable. Think of it as a row of boxes, each holding a value.

Example: Declaring Arrays

```
fn main(x : Field, y : Field) {
        let my_arr = [x, y];
        let your_arr: [Field; 2] = [x, y]; ✓
}
```

Explanation:

- `my_arr` is an array with two elements, x and y.
- `your_arr` is the same, but we explicitly say it is an array of two Field values.
- Both ways are valid.

Accessing Array Elements

You can get a value from an array by its position (called an ”index”). The first element is at index 0.

```
fn main() {
        let a = [1, 2, 3, 4, 5];

        let first = a[0];
        let second = a[1]; ✓
}
```

Explanation:

- a[0] gets the first value (1).
- a[1] gets the second value (2).
- Indexing starts at 0.

Accessing Array Elements with Output

You can print array values to confirm what you're accessing.

```
fn main() {
        let a = [1, 2, 3, 4, 5,10,23];

        let first = a[0];
        let second = a[1];
        let sixth = a[5];
        let seventh = a[6];

        println("First␣value:");
        println(first);

        println("Second␣value:");
        println(second);

        println("Sixth␣value:");
        println(sixth);

        println("Seventh␣value:");
        println(seventh);
}
```

Explanation:

- `println("text")` prints a label.
- `println(value)` prints the actual value.
- Noir prints numbers in hexadecimal format (e.g., `0x01`).

Add Prover.toml

```
inputs = { x = 1, y = 2 }
```

Output When Program Runs

When you run `nargo execute`, this is the output:

```
First value:
0x01
Second value:
0x02
```

Interpretation:

- The array value at index 0 is 1 (shown as `0x01`).
- The array value at index 1 is 2 (shown as `0x02`).
- This confirms your indexing logic works.

Testing Array Access in Noir

Passing Test:

```
#[test]
fn test_array_access_pass() {
        let a = [1, 2, 3, 4];
        let x = a[0];                      ✓
        assert(x == 1); // This passes
}
```

Failing Test:

```rust
// This test is expected to fail.
#[test(should_fail)]
fn test_array_access_fail() {
        let a = [1, 2, 3, 4];
        let x = a[1];
        assert(x == 99); // This fails
}
```

Mutating Arrays

If you make an array mutable, you can change its values after creating it.

```
fn main() {
let mut arr = [1, 2, 3, 4, 5];  // mutable array
assert(arr[0] == 1);       // initial value is 1
arr[0] = 42;  // mutate: change value at index 0
assert(arr[0] == 42); // confirm new value is 42
}
```

Explanation:

- Noir enforces immutability by default, which is good for avoiding unintended side effects.

- `mut` is required for in-place updates, like assigning `arr[0] = 42`.
- `let mut arr` makes the array changeable.
- `arr[0] = 42` changes the first value.
- `assert` checks expected values.
- Using `assert()` in such examples is not just for testing — it confirms correctness and helps catch bugs early.

Mutation Without mut: Causes Error

Common Mistake: Forgetting `mut`

This code will cause a compile-time error:

```
fn main() {
let arr = [1, 2, 3];
arr[0] = 99; //  Error: Cannot assign to
    immutable variable
                   }
```

Why?

- Arrays are **immutable by default**.
- If you want to modify the array, declare it with `mut`.

Correct Version:

```rust
let mut arr = [1, 2, 3];
arr[0] = 99; // Works now
```

Mutating Arrays and Printing Values

Changing Values and Observing Effects

Arrays in Noir (like in Rust) are **mutable** if declared with `mut`.

When mutable, you can overwrite any existing element with any new value of the same type — it does not have to be one of the original values.

Example:

```
fn main() {
        let mut arr = [1, 2, 3];  // Original:
           [1, 2, 3]
        arr[0] = 99;              // Now becomes
           : [99, 2, 3]

        println("Index␣0␣value:");
        println(arr[0]); // Prints 99

        println("Index␣2␣value:");
        println(arr[2]); // Prints 3
}
```

Note: Attempting `arr[3]` would cause an error — index out of bounds.

Initializing Arrays with Repeated Values

You can quickly make an array where every element is the same value.

```
let array: [Field; 32] = [0; 32];
```

The slide introduces a concise way of creating an array filled with the same value repeated multiple times. In this case, you're using 0.

Code Breakdown:

```
let array: [Field; 32] = [0; 32];
```

What it means:

- `Field`: The type of each element in the array (Noir's core scalar type).
- `[Field; 32]`: A fixed-size array of 32 `Field` elements.
- `[0; 32]`: Syntax meaning "fill the array with the value 0, repeated 32 times."

The result is an array like:

```
[0, 0, 0, ..., 0] // repeated 32 times
```

Useful for:

- Initializing a buffer
- Creating a uniform starting state
- Allocating space before updating values later

Accessing Values in Repeated Arrays

Verify Array Initialization

You can check individual values in the initialized array to confirm they are set as expected.

```
fn main() {
        let array: [Field; 32] = [0; 32];

        println("First␣value:");
        println(array[0]);  // Prints 0

        println("Last␣value:");
        println(array[31]); // Also prints 0
}
```

Note: All 32 elements are set to 0. Indexing goes from 0 to 31.

Modifying a Repeated Array

Change Specific Elements

Although the array is initialized with repeated values, individual elements can still be changed.

```
fn main() {
        let mut array: [Field; 32] = [0; 32];
        array[5] = 99;

        println("Value␣at␣index␣5:");
        println(array[5]);  // Prints 99

        println("Value␣at␣index␣6:");
        println(array[6]);  // Still 0
}
```

Note: Only the specified index is changed; others remain untouched.

Multidimensional Arrays

Arrays can contain other arrays, making a grid or table of values.

```
let array : [[Field; 2]; 2];
let element = array[0][0];
```

This slide introduces the concept of **multidimensional arrays** in Noir. Just like in other languages such as Rust or C, Noir allows you to create arrays that contain other arrays.

What it means:

- `[[Field; 2]; 2]`: This declares a 2×2 matrix — an array with 2 rows, each containing 2 Field elements.
- `array[0][0]`: This accesses the first element in the first row.

Visual Representation:

```
Row 0 $\rightarrow$ [x, y]
Row 1 $\rightarrow$ [a, b]
```

So `array[0][0]` gives `x` (first row, first column).

Initializing a 2D Array

Working with 2D Arrays

You can declare and initialize a multidimensional array with values.

```
fn main() {
        let array: [[Field; 2]; 2] = [[1, 2],
            [3, 4]];

        let top_left = array[0][0];    // 1
        let bottom_right = array[1][1]; // 4
}
```

Explanation:

- The array has 2 rows, each with 2 columns.
- `array[0][0]` accesses the first element (top-left).
- `array[1][1]` accesses the last element (bottom-right).

Printing from a 2D Array

Displaying Values

You can print individual elements of a multidimensional array.

```
fn main() {
        let array: [[Field; 2]; 2] = [[10, 20],
            [30, 40]];

        println("Top␣left:");
        println(array[0][0]); // 10

        println("Top␣right:");
        println(array[0][1]); // 20

        println("Bottom␣left:");
        println(array[1][0]); // 30

        println("Bottom␣right:");
        println(array[1][1]); // 40
}
```

Note: Each element is accessed using two indices: `array[row][column]`.

Mutating and Printing 2D Arrays

Update and Display Grid Values

You can modify individual elements in a multidimensional array using indexing and then print them to confirm the changes.

```
fn main() {
let mut array: [[Field; 2]; 2] = [[1, 2], [3,
    4]];

        // Mutate specific elements
        array[0][1] = 20; // change top right
            value
        array[1][0] = 30; // change bottom left
            value
        // Print all elements
        println("Top␣Left:");
        println(array[0][0]); // 1

        println("Top␣Right:");
        println(array[0][1]); // 20

        println("Bottom␣Left:");
        println(array[1][0]); // 30

        println("Bottom␣Right:");
        println(array[1][1]); // 4
}
```

Explanation:

- You must use `mut` to make the array mutable.
- Access and update values using `array[row][column]`.
- `println` helps verify the updated values.

Array Methods: `len`, `sort`, `map`, `fold`, `concat`

a. len - Get Array Length

```
fn main() {
        let array = [42, 42];
        assert(array.len() == 2)
            ;
}
```

b. sort - Sort an Array

```
fn main() {
        let arr = [42, 32];
        let sorted = arr.sort();
        assert(sorted == [32,
            42]);
}
```

Array Methods continued

c. map - Apply a Function to Each Element

```
let a = [1, 2, 3];
let b = a.map(|a| a * 2); // b
    is now [2, 4, 6]
```

d. fold - Reduce Array to Single Value

```
fn main() {
        let arr = [2, 2, 2, 2,
            2];
        let folded = arr.fold(0,
             |a, b| a + b);
        assert(folded == 10);
}
```

e. concat - Concatenate Two Arrays

```
fn main() {
```

Dynamic Indexing

```
fn main(x: u32) {
        let array = [1, 2, 3,
            4];
        let _b = array[x]; // x
            must be 0, 1, 2, or 3
}
```

Explanation:

- x is used as index.
- If x is out of range, it will cause an error.

Hands-On Activity 1: Create and Mutate an Array

```
fn main() {
        let mut arr = [10, 20,
            30, 40, 50];
        arr[2] = 99;
        arr.for_each(|x| {
                println(f"{x}");
        });
}
```

Explanation:

- We change the third value to 99.
- `for_each` prints each value.

Activity 2: Use map to Transform an Array

Task: Double each value in an array.

```
let a = [1, 2, 3];
let b = a.map(|a| a * 2); // b
   is now [2, 4, 6]
```

Activity 3: Fold an Array

Task: Add up all the values in an array.

```
fn main() {
        let arr = [5, 10, 15];
        let sum = arr.fold(0, |a
            , b| a + b);
        assert(sum == 30);
}
```

Best Practices

- All elements in an array must be the same type.
- Arrays have a fixed size.
- Use array methods for common tasks.
- Use constant indices when possible for safety.

What is an Array of Structs?

An array of structs is an array where each element is a struct.

Example:

```
struct Animal {
        hands: Field,
        legs: Field,
        eyes: u8,
}

fn main() {
        let dog = Animal { eyes:
            2, hands: 0, legs: 4
            };
        let cat = Animal { eyes:
            2, hands: 0, legs: 4
            };
        let animals: [Animal; 2]
```

Nested Arrays and Structs

More Complex Example

```
struct InnerStruct {
        small_array: [u32; 2],
        big_array: [u32; 5],
}

struct MyStruct {
        x: u32,
        y: u32,
        z: u32,
        nested_struct:
            InnerStruct,
}

fn main() {
        let nested_struct =
```

Key Points

- Arrays of structs work like arrays of primitive types.
- Use dot notation to access struct fields after indexing.
- This enables organizing and processing complex data.