WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in noir

Lecture 111

July 9, 2025

Introduction to Vectors in Noir

What are Vectors?

Vectors, represented as `Vec<T>` in Noir, are collections that can grow or shrink in size as your program runs. This is different from regular arrays, which have a size that must be fixed at compile time.

- **Dynamic:** You can add or remove elements at runtime.
- **Generic:** The `T` in `Vec<T>` means you can have vectors of any type.

Why use vectors?

Vectors give you flexibility when you don't know in advance how many elements you'll need.

Note: These vectors are not the same as vectors in mathematics (like geometric or linear algebra vectors). In programming, a `Vec<T>` is a flexible, resizable collection of values.

Concept	**Vectors in Mathematics**	**`Vec<T>` in Noir (and Rust)**
Meaning	Ordered sequence of numbers, representing direction and magnitude (e.g., in physics or linear algebra).	Dynamic collection (list) of any type of data—like an expandable array.
Structure	Fixed size (e.g., 3D vector = always 3 components).	Flexible size: can grow or shrink.
Operations	Dot product, cross product, linear transformations.	Push, pop, iterate, access elements.
Field of Use	Geometry, physics, cryptography (mathematical sense).	Programming data structures (software engineering).

Declaring and Initializing a `Vec<T>`

How do you create a new vector?

Creates an empty vector that holds bytes:

```
let mut v: Vec<u8> = Vec::new();
```

- `Vec<u8>` means the vector will store bytes.
- `mut` allows the vector to grow or shrink later.
- Without `mut`, the vector would be immutable.

How do you create a vector with initial values?

Creates a vector pre-filled with values:

```
let mut v = Vec::from([1, 2, 3]);
```

- This uses `Vec::from` to convert an array into a vector.
- The type `Vec<u8>` is automatically inferred from the array.

Why is mutability important?

- Vectors are resizable only if declared with `mut`.
- Without `mut`, you cannot use `push` or `pop`.
- This is crucial when the number of elements may change during execution.

Common `Vec<T>` Operations

Adding Elements: `push`

Adds a value to the end of the vector:

```
v.push(42);
```

- Increases the vector length by one.
- Example: ‘[1, 2]‘ becomes ‘[1, 2, 42]‘.

Removing Elements: `pop`

Removes and returns the last value:

```
let last = v.pop();
```

- Returns `Some(value)` if not empty, otherwise `None`.
- Reduces the vector size by one.

Checking Length: `len`

Returns the number of elements:

```
let length = v.len();
```

- No need to loop manually—just call `len()`.

Accessing Elements:

Read an element by index:

```
let first = v[0];
```

- Indexing starts from zero.

- Panics if you access beyond current length.

Adding Elements: `push`

Below is an example of how to add elements to a vector (`Vec<T>`) in Noir:

```
fn add_element() {
        let mut v: Vec<u32> = Vec::new(); //
            Create a new, empty Vec

        v.push(42); // Add the value 42 to the
            end of the vector
        // v is now [42]
}
```

Removing Elements: `pop`

Example of how to remove the last element from a vector:

```
fn remove_element() {
    let mut v = Vec::from([10, 20, 30]);
        let last = v.pop(); // Removes and
            returns the last element
        // last is Some(30), v is now [10, 20]
}
```

Checking Length: `len`

Example showing how to check the length of a vector:

```
fn check_length() {
        let v = Vec::from([1, 2, 3, 4]);
        let length = v.len(); // length is 4
}
```

Accessing Elements by Index

Example of how to access an element by index:

```
fn access_element() {
        let v = Vec::from([7, 8,
            9]);
        let first = v[0]; //
            first is 7
}
```

Important Notes:

- `push` and `pop` require the vector to be mutable (`mut`).
- Accessing an index beyond the vector's length (e.g., `v[100]` when length is less) will cause a runtime panic.

Common `Vec<T>` Operations in Noir

Demonstrating `push`, `pop`, `len`, indexing, and sum:

```
// Noir example: Demonstrating common Vec<T>
    operations

// Noir: Vec<T> operations demo

fn main() {
        // Create a new mutable vector of Field
            elements
        let mut v: Vec<Field> = Vec::new();

        // Add elements to the vector
        v.push(10);
```

```
    v.push(20);
    v.push(30);

    // Length after push
    let length_after_push = v.len();
    assert(length_after_push == 3);

    // Pop the last element (returns Field,
       not Option)
    let last = v.pop();
    assert(last == 30);

    // Length after pop
    let length_after_pop = v.len();
    assert(length_after_pop == 2);
```

```
        // Safely access elements using .get(),
            which returns a Field
        assert(v.len() == 2);
        let first = v.get(0);
        let second = v.get(1);
        assert(first == 10);
        assert(second == 20);

        // Sum the remaining elements
        let sum = first + second;
        assert(sum == 30);
}
```

Example: Vectors and Public Outputs in Noir

Noir Circuit Example:

```
fn main() -> pub (Field, Field) {
        let mut v: Vec<Field> = Vec::new();
        v.push(10);
        v.push(20);
        v.push(30);
        let _ = v.pop()
        let first = v.get(0);
        let second = v.get(1);
        let sum = first + second;
        (first, sum)
}
```

nargo execute Output:

```
        Circuit output: Vec([Field(10), Field
            (30)])
```

Explanation of the Output

- **First output:** `Field(10)` → Value of `first` (v[0])
- **Second output:** `Field(30)` → Sum of `first + second` (10 + 20)

Key Takeaways:

- What you **return** is what appears in the circuit output.
- Use `pub` in the return type to expose public outputs.
- Vectors (`Vec<Field>`) allow dynamic manipulation inside Noir circuits.

Example: Demonstrating Multiple Vec Operations in Noir

Circuit Code:

```
fn main() -> pub (Field, Field, Field, Field,
    Field) {
        let mut v: Vec<Field> = Vec::new();
        v.push(10);
        let pushed1 = v.get(0);
        v.push(20);
        let pushed2 = v.get(1);
        v.push(30);
        let length_after_push: Field = v.len().
            into();
        let popped = v.pop();
```

```
        let length_after_pop: Field = v.len().
            into();
        (pushed1, pushed2, popped,
            length_after_push, length_after_pop)
}
```

Example – Summing Elements

```
let mut v = Vec::from([1, 2, 3, 4, 5]);
let mut sum = 0;
for i in 0..v.len() {
        sum += v[i];
}
```

Result: sum = 15

Example – Dynamic Resizing

```
let mut v: Vec<u8> = Vec::new();
v.push(10);
v.push(20);
v.push(30);
let last = v.pop();
```

Use Cases: Buffers, stacks, user input lists, etc.

Summary

- Vectors (`Vec<T>`) in Noir are dynamic, resizable collections.
- Key operations: `push`, `pop`, `len`, indexing.
- Prefer vectors when size flexibility is needed.

Further Exploration

- Write functions that take `Vec<T>` as input.
- Combine vectors with structs.
- Handle edge cases (empty vectors, large inputs).

Questions? Feel free to ask!