

# Slices | Arrays & Slices

# ■ Arrays

- ◆ Contiguous memory region
- ◆ All elements have the same size
- ◆ Arrays are not dynamically sized
  - Size must be *hard-coded*
  - Usually prefer **Vector**
- ◆ Useful when writing algorithms with a fixed buffer size
  - Networking, crypto, matrices

# ■ Syntax

```
let numbers = [1, 2, 3];  
let numbers: [u8; 3] = [1, 2, 3];
```

                    ↑          ↑  
                    Type      Element Count

```
fn func(arr: [u8; 3]) {}  
fn func(arr: &[u8]) {}  
fn func(arr: &mut [u8]) {}
```

# ■ Slices

- ◆ A borrowed *view* into an array
- ◆ Can be iterated upon
- ◆ Optionally mutable
- ◆ Indices bounded by the slice size
  - Cannot go out of bounds of the initial slice
- ◆ Can create any number of subslices from an existing slice

# ■ Slices – View Into An Array

[char; 10]

Array    0    1    2    3    4    5    6    7    8    9



Slice                    0    1    2    3

&[char]

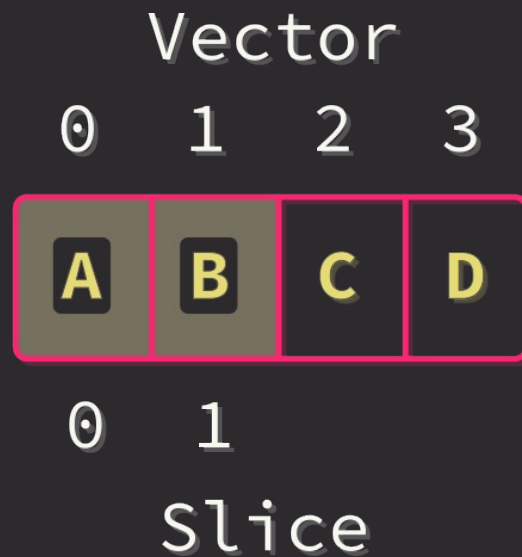
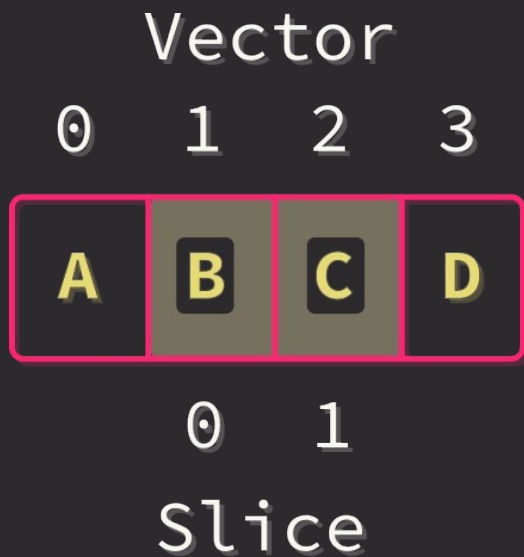
# ■ Slices & Vectors

- ◆ Borrowing a Vector as an argument to a function that requires a slice will automatically obtain a slice
- ◆ Always prefer to borrow a slice instead of a Vector

```
fn func(slice: &[u8]) {}  
  
let numbers = vec![1, 2, 3];  
func(&numbers);  
let numbers: &[u8] = numbers.as_slice();
```

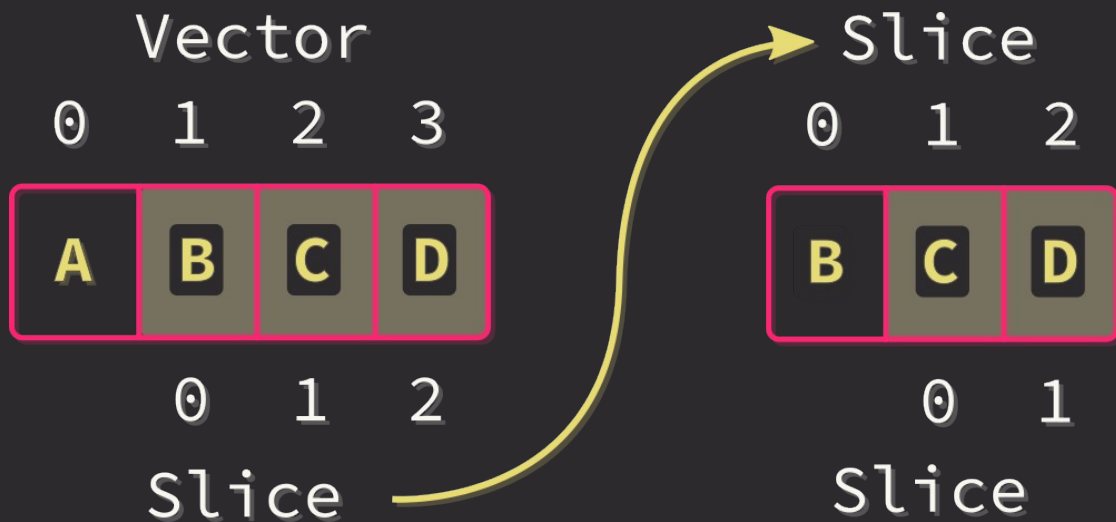
# ■ Slicing With Ranges

```
let chars = vec!['A', 'B', 'C', 'D'];  
let bc = &chars[1..=2];  
let ab = &chars[0..2];
```



# Subslices

```
let chars = vec!['A', 'B', 'C', 'D'];  
let bcd = &chars[1..=3];  
let cd = &bcd[1..=2];
```





# ■ Recap

- ◆ Arrays must be statically initialized with hard-coded lengths
- ◆ Slices are a way to access parts of an array
- ◆ Array-backed data structures like Vectors can be sliced
- ◆ Slice lengths are always bound by the size of the slice
- ◆ Subslices can be created from existing slices