**Arrays**

**Definition**:

* An **array** is a **fixed-length collection** of elements, all of the same type.
* These elements are **indexable** and stored in **contiguous memory locations**.

**Syntax**:

* To declare an array in Go, use square brackets [] with a number (indicating the array's fixed length), followed by the type of elements in the array.
  + Example: var arr [3]int32 declares an array of 3 elements, all of type int32.
* The **length** of the array is **fixed** and cannot change after initialization.

**Default Values**:

* When an array is declared, Go automatically initializes all elements to their **default values**.
* For int32, the default value is 0, so an array like var arr [3]int32 will contain [0, 0, 0].

**Indexing**:

* Arrays are **zero-indexed**, meaning the first element is at index 0, the second at index 1, and so on.
* You can access elements using their index: arr[0] to get the first element, and so on.
* You can also modify elements using indexing, e.g., arr[0] = 5 to change the first element.

**Memory Allocation**:

* Arrays store elements in **contiguous memory locations**.
  + For example, an int32 array of size 3 will take up 3 \* 4 bytes = 12 bytes (since each int32 takes 4 bytes).
* The Go compiler doesn't need to track each element's memory location individually. Instead, it stores the **address of the first element** and increments by the size of the type (4 bytes for int32) to access subsequent elements.

**Memory Location Access**:

* You can print the memory location of each element using the & (address-of) operator, e.g., &arr[0] to see the memory location of the first element.

**Array Initialization**:

* Arrays can be initialized using different syntaxes:
  + var arr [3]int32 = [3]int32{0, 0, 0} (explicitly setting values).
  + arr := [3]int32{1, 2, 3} (shorthand initialization).
  + You can even omit the size and let the compiler infer it: arr := [...]int32{1, 2, 3}.

**Slices**

**1. Definition:**

* According to the Go documentation, **slices** are just **wrappers around arrays**.
* **Under the hood**, a slice is essentially an array with some additional functionality.

**2. Creating a Slice:**

* To create a slice, you **omit the length value** when initializing an array, like so:
* var slice1 []int32 = []int32{4, 5, 6}
* This creates a **slice** rather than a fixed-size array.

**3. Appending to a Slice:**

* Slices, unlike arrays, can grow dynamically. You can add values to a slice using the built-in append function:
* slice1 = append(slice1, 7)
* The append function takes two parameters:
  + The **slice** you want to append to.
  + The **value** to append to the slice.
* The function then returns a **new slice** that includes the appended element.

**4. Underlying Array and Reallocation:**

**What happens under the hood?**

* Initially, an **array** is allocated that can hold a specific number of elements (e.g., 3 elements).
* When you try to append a new value to the slice, Go checks if the underlying array has enough room to store the new element.
* If the array **doesn't have enough capacity**, Go will:
  + Create a **new array** with enough capacity.
  + **Copy** the existing values from the old array to the new one.
  + Append the new value to the new array.
* This results in a **new array** being allocated, which means the slice now points to a **different memory location**.

**5. Capacity vs. Length:**

* **Length** of a slice refers to the number of elements it currently holds.
* **Capacity** refers to the maximum number of elements the slice can hold before the underlying array needs to be reallocated.
* Initially, the **length** and **capacity** of the slice are the same. After appending, the **length** grows, but the **capacity** may increase as well to accommodate future growth.
* For example, after appending:
  + The length might be 4 (since 4 elements are in the slice).
  + The capacity might be 6, meaning the underlying array has room for 6 elements, even though the slice only has 4 elements at the moment.

**6. Accessing Out-of-Bounds Elements:**

* If you try to access an index that’s out of range for the slice (for example, index 6 in a slice with only 4 elements), you'll get an **index out of range error**.
* This is because the slice’s **length** determines the accessible range of indices.

**7. Appending Multiple Values:**

* You can also append **multiple values** to a slice at once using the **spread operator**:
* slice1 = append(slice1, slice2...)

**8. Creating a Slice with make:**

* You can create a slice using the built-in make function:
* var slice3 []int32 = make([]int32, 3, 8)
* // creates a slice of length 3 and capacity 8, [0,0,0]
* The make function allows you to:
  1. Specify the **length** of the slice.
  2. Optionally specify the **capacity** of the slice.
* **Why specify capacity?**
  1. By default, the **capacity** of the slice will be equal to its **length**.
  2. If you know that you’re going to need a large number of elements, specifying the capacity upfront can help improve performance. This avoids the need for Go to reallocate and copy the array multiple times as the slice grows.

**9. Performance Consideration:**

* Reallocating and copying the underlying array when appending elements can **slow down performance**, especially when you need to resize the array multiple times.
* By specifying the slice's **capacity** in advance (using make), you can avoid unnecessary reallocations, improving the efficiency of your program.

**10. Summary:**

* **Slices** are flexible and dynamic, allowing you to append elements.
* The underlying array may need to be reallocated if the slice exceeds its initial capacity.
* You can use make to create a slice with a specified length and capacity to optimize performance.
* Understanding **capacity** and **length** helps manage memory and avoid performance hits due to frequent reallocations.

**Maps**

### 1. ****Definition****:

A **map** is a collection of **key-value pairs**, where you can **look up a value by its key**. It’s a useful data structure for efficiently storing and retrieving data based on a unique identifier (the key).

### 2. ****Creating a Map****:

* You can create a map using the built-in make function:
* var map1 map[string]uint8 = make(map[string]uint8)

This creates a map where:

* + The **keys** are of type string.
  + The **values** are of type uint8 (unsigned 8-bit integer).
* Alternatively, you can initialize a map with values immediately:
* var map3 = map[string]uint8{"Adam": 23, "Bob": 24, "Jason": 25}

This creates a map where the keys represent people's names and the values represent their ages.

### 3. ****Accessing Map Values****:

* You can retrieve a value from a map by using its **key**:
* fmt.Println(map3["Adam"])
* However, if the key does not exist in the map, Go will return the **default value** for the map's value type:
  + In this case, since the values are uint8, the default value would be 0.

### 4. ****Checking for Key Existence****:

* If you try to access a key that doesn’t exist, you might want to check whether the key is actually present in the map.
* Go provides a **second return value** when accessing a map:

var age, ok = map3["Emma"]

    if ok {

        fmt.Printf("The age is %v\n", age)

    } else {

        fmt.Printf("Invalid\n")

    }

* + age: the value for the key (or the default value if the key doesn’t exist).
  + exists: a **boolean** that indicates whether the key exists in the map (true if the key exists, false otherwise).
* This way, you can handle the case where the key doesn't exist without mistakenly using the default value.

### 5. ****Deleting Map Entries****:

* To **delete** a key-value pair from a map, Go provides the delete function:
* delete(map3, "Adam")
  + The first argument is the map.
  + The second argument is the key you want to delete.
* The deletion is done **by reference**, meaning the map is modified in place, and no return value is provided.

### 6. ****Iterating Over a Map****:

* You can **iterate over maps** using a **for loop** with the range keyword:

for name, age := range map3 {

    fmt.Printf("Name: %v Age: %v\n", name, age)

}

* In the loop:
  + key is the current key in the map.
  + value is the corresponding value for that key.
* **Note**: Maps in Go do not preserve the **order** of keys. When iterating over a map, the order in which you get the keys can vary each time you run the loop.

### 7. ****Iterating Over Arrays and Slices****:

Similar to maps, you can also iterate over **arrays** and **slices** using the range keyword:

for index, value := range arr2 {

    fmt.Printf("Index: %v Value: %v\n", index, value)

}

### 8. ****Go Loops (No**** while ****Loop)****:

* Go doesn’t have a while loop, but you can achieve similar functionality with the for loop:

var i int = 0

    for i < 10 {

        fmt.Println(i)

        i = i + 1

    }

This acts like a while loop, continuing until the condition (i < 10) is false.

* Alternatively, you can use a for loop without any condition (infinite loop) and break it when needed:

var j int = 10

    for {

        if j < 0 {

            break

        }

        fmt.Println(j)

        j = j - 1

    }

* You can also have a for loop with three parts (initialization, condition, post):

for k := 0; k < 5; k++ {

        fmt.Println(k)

    }

### Summary of ****Maps****:

* **Map**: A collection of key-value pairs.
* **Creation**: Can be created using make or initialized with values.
* **Access**: You can retrieve values using keys, and handle cases where keys don't exist using the second return value.
* **Deletion**: delete function removes key-value pairs.
* **Iteration**: Maps don’t preserve key order. Use range to loop through keys and values.
* **Key Existence**: Use the second return value from a map access to check whether a key exists.

Maps are useful for efficiently looking up values associated with keys, and the additional features like checking key existence and deleting entries provide flexibility in handling dynamic data structures.