Arrays

Data Structures and Algorithms (DSA).
•**Data Structures** are structures used in programming code to **store data**. Data is considered the "fuel" for development. Different types of data (linear, hierarchical, etc.) are stored in different data structures.
•**Algorithms** are ways to perform **efficient operations** on data stored in data structures. Examples include searching for data or sorting data.
The need for arrays arises from the problem of storing multiple pieces of similar data, like the marks of many students or data for many users. Creating and tracking a large number of individual variables for this purpose is **very difficult and hectic**.

Arrays are introduced as a data structure to **resolve the multiple variable problem**.
•Arrays can be visualised as a **block or structure** capable of storing multiple pieces of information under a single name.
•This structure has a **single name** (a single variable) that needs to be tracked, regardless of the number of elements it holds.

Arrays have several key properties:
•They can store data of the **same type** (e.g., all integers, all doubles). If one block stores an integer, all blocks will store integers.
•They are **contiguous in memory**, meaning the data is stored together sequentially. If the first element starts at memory address 100 (assuming 4 bytes per element), the next will start at 104, then 108, and so on. This sequential arrangement helps in calculating the address of other elements from the starting address.
•They are **linear**, which means the data can be imagined as arranged in a straight line, one after another....

Creating (Declaring) and Initialising Arrays:
•To create an array, you need to specify the **data type** of the elements, the **array name**, and its **size** within square brackets. For example: `int marks;` creates an integer array named `marks` with a size of 5.
•The size specifies the total number of "blocks" or elements the array can hold. Each block occupies memory equal to the size of its data type.
•Arrays can be initialised with values when created using curly braces `{}`. For example: `int marks[] = {99, 100, 54, 36, 88};`.
•When initialising with values, you can **skip the size** in the square brackets if the number of initial values matches the desired size. The compiler will automatically determine the size based on the number of elements provided.
•If you provide a size that is larger than the number of initial values, the array will be created with the specified larger size, and the provided values will fill the beginning elements.

Accessing Array Elements:
•Data within an array is accessed using an **array index** or position.
•Array indexes **start from 0**. This is why loops are often started from 0 when processing arrays.
•The valid indexes for an array range from `0` up to `size - 1`. For a size 5 array, indexes are 0, 1, 2, 3, 4.
•To access an element, use the array name followed by the index in square brackets, e.g., `marks` accesses the element at index 0.
•`array_name[index]` can be treated like a **single variable**; its value can be printed (`cout`), changed, or input (`cin`)....
•Attempting to access an index **outside the valid range** (0 to size - 1), such as `size` or -1, will result in a warning or potentially incorrect (garbage) values.

Looping through Arrays:
•Loops are commonly used to **perform operations on all or many elements** of an array efficiently, especially for large arrays.
•A typical loop iterates from index `0` up to `size - 1`. Example: `for (int i = 0; i < size; i++) { /* access or process array[i] */ }`.
•The **size of the array** can be calculated dynamically using `sizeof(array_name) / sizeof(array_element_type)`. For an integer array `marks`, this would be `sizeof(marks) / sizeof(int)`. However, the size is often available in a separate variable.
•Loops can be used to **input values** into an array (`cin`) or **output values** from an array (`cout`).

Finding Minimum and Maximum Values in an Array:
•This involves iterating through the array using a loop and comparing elements.

•To find the **smallest** value:
◦Initialize a variable, e.g., `smallest`, with the **largest possible integer value** (`INT_MAX` in C++).
◦Loop through the array. For each element `nums[i]`, compare it with the current `smallest` value.
◦If `nums[i]` is **less than** `smallest`, update `smallest` to `nums[i]`.
◦After the loop, `smallest` will hold the minimum value found.

•To find the **largest** value:
◦Initialize a variable, e.g., `largest`, with the **smallest possible integer value** (`INT_MIN` in C++).
◦Loop through the array. For each element `nums[i]`, compare it with the current `largest` value.
◦If `nums[i]` is **greater than** `largest`, update `largest` to `nums[i]`.
◦After the loop, `largest` will hold the maximum value found.

•C++ provides built-in `min()` and `max()` functions that can simplify the comparison and update step.
•Both the smallest and largest values can be found within a **single loop** iterating through the array once.

Pass by Reference with Arrays:
•This concept contrasts with "Pass by Value", where a copy of a primitive variable (like a single integer) is passed to a function, and changes inside the function don't affect the original variable.
•Arrays are considered **non-primitive data types**.
•When an array is passed as an argument to a function in C++, it is **implicitly passed by reference**.
•Instead of copying the entire array, the function receives the **starting memory address** (or a reference) of the original array.... The array name itself acts as a pointer to this starting address.
•Because the function operates on the original data's address, any **changes made to the array elements inside the function *will* affect the original array** in the calling function (like `main`).

Linear Search Algorithm:
•Linear Search is an **algorithm** used to find the index of a specific **target value** within an array.
•The process is very simple: iterate through the array starting from the first element (index 0).
•In each step, **compare** the current array element with the target value.
•If a match is found, **return the index** of the current element.
•If the loop finishes without finding the target value after checking all elements, **return a special value** (commonly -1) to indicate that the target was **not found** in the array. -1 is chosen because it is an invalid index for any array.
•It is described as one of the simplest algorithms.•
The implementation typically uses a `for` loop iterating from `0` to `size - 1` and an `if` statement inside the loop to check for the target.
•The **time complexity** of Linear Search is **O(n)** (linear time), meaning the time taken increases linearly with the size of the array. This is the origin of the name "Linear Search".

Reversing an Array:
•The goal is to **reverse the order of elements** in an array **within the original array**, without creating a new one.
•A common and important approach for this is the **Two-Pointer Approach**.
•This approach uses two variables (pointers tracking index): `start`, initialised to the first index (0), and `end`, initialised to the last index (`size - 1`).
•A loop runs as long as the `start` index is **less than** the `end` index (`while (start < end)`).

•Inside the loop:
◦The elements at the `start` index and the `end` index are **swapped**..C++ provides a `swap` function for this.
◦The `start` index is **incremented** (`start++`)
◦The `end` index is **decremented** (`end--`)

•This process continues, swapping elements from the outside inwards, until the pointers meet or cross. When `start` is no longer less than `end` (`start >= end`), the array is fully reversed, and the loop terminates
•The Two-Pointer Approach is a significant concept used in many other problems and algorithms beyond just array reversal.
•The **time complexity** for reversing an array using this approach is **O(n)**.