ARRAYS and VECTORS ;

Arrays:

Uses of Arrays

- To store and manage collections of data of the same type.
- Efficient access and modification of elements using indexing.
- Commonly used in algorithms like sorting, searching, and storing large datasets.

What Are Arrays in C++?

An **array** is a collection of elements of the same data type stored in contiguous memory locations. It is used to store multiple values in a single variable, rather than declaring separate variables for each value.

Key Characteristics of Arrays

1. All elements in an array must be of the same type.
2. Array indexing starts at **0**.
3. Arrays have a fixed size determined at the time of declaration.

Why Use Arrays?

Arrays are used for managing and organizing data efficiently. They allow:

1. **Storage of Multiple Values**: Arrays allow storing multiple elements of the same type without needing to declare multiple variables.
2. **Efficient Access and Modification**: Elements can be accessed and updated using their index in **O(1)** time complexity.
3. **Support for Algorithms**: Arrays are fundamental for algorithms such as **sorting**, **searching**, and **matrix operations**.

4. **Contiguous Memory Allocation**: Makes arrays efficient in terms of memory usage and access.

Features of Arrays

Below are the features, their explanations, and how to perform the corresponding operations with code:

1. Homogeneous Data

- All elements in an array are of the same data type.
- Example:

```
int arr[5] = {1, 2, 3, 4, 5}; // Array of integers
float scores[3] = {98.5, 88.3, 76.2}; // Array of floats
```

---

2. Contiguous Memory Allocation

- Array elements are stored in adjacent memory locations.
- Example:

```
#include <iostream>using namespace std;

int main() {
    int arr[3] = {10, 20, 30};

    // Print memory addresses of array elements
    for (int i = 0; i < 3; i++) {
        cout << "Address of arr[" << i << "]: " << &arr[i]
<< endl;
    }
    return 0;
```

```
}
```

Output:

```
Address of arr[0]: 0x7ffee4b40d70
Address of arr[1]: 0x7ffee4b40d74
Address of arr[2]: 0x7ffee4b40d78
```

Here, each element is 4 bytes apart for an `int`.

---

3. Fixed Size

- Once declared, the size of an array cannot change.
- Example:

```
cpp
Copy code
int arr[5]; // Array of size 5
// arr = new int[10]; // This is NOT allowed in C++.
```

---

4. Zero-Based Indexing

- Array indices start at `0`.
- Example:

```
cpp
Copy code
int arr[3] = {10, 20, 30};
cout << arr[0]; // Outputs 10 (1st element)
cout << arr[1]; // Outputs 20 (2nd element)
```

---

5. Random Access

- You can directly access any element using its index.
- Example:

```
cpp
Copy code
#include <iostream>using namespace std;

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    cout << "Before Modification: " << arr[2] << endl;

    // Modify element at index 2
    arr[2] = 10;
    cout << "After Modification: " << arr[2] << endl;
    return 0;
}
```

Output:

```
mathematica
Copy code
Before Modification: 3
After Modification: 10
```

---

7. Supports Multiple Dimensions

- Arrays can have one, two, or more dimensions.
- Example:

```
cpp
Copy code
```

```
#include <iostream>using namespace std;

int main() {
    int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };

    // Access elements
    cout << "arr[0][1]: " << arr[0][1] << endl; // Outputs
2
    cout << "arr[1][2]: " << arr[1][2] << endl; // Outputs
6

    return 0;
}
```

Output:

```
less
Copy code
arr[0][1]: 2
arr[1][2]: 6
```

Operations on Arrays

1. How to Declare and Initialize an Array

```
int arr[5] = {1, 2, 3, 4, 5}; // Declaration and Initializati
on
```

2. How to Access Array Elements

```
cout << arr[0]; // Access the first element
cout << arr[3]; // Access the fourth element
```

3. How to Modify Array Elements

```
arr[1] = 50; // Changes the second element to 50
```

Code Example for Key Features and Operations

```
#include <iostream>using namespace std;

int main() {
    // Declare and initialize an array
    int arr[5] = {10, 20, 30, 40, 50};

    // Access elements using index
    cout << "Element at index 0: " << arr[0] << endl; // Outp
uts 10
    cout << "Element at index 3: " << arr[3] << endl; // Outp
uts 40

    // Modify elements
    arr[2] = 35; // Update 3rd element
    cout << "Updated element at index 2: " << arr[2] << endl;

    // Print all elements
    cout << "All elements: ";
    for (int i = 0; i < 5; i++) {
```

```
        cout << arr[i] << " ";
    }
    cout << endl;

    // Example of contiguous memory
    cout << "Memory addresses of elements:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }

    // Example of garbage value (uninitialized local array)
    int uninitializedArr[3];
    cout << "Garbage values in uninitialized array: ";
    for (int i = 0; i < 3; i++) {
        cout << uninitializedArr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

1. Searching Techniques

Arrays can be searched using different algorithms depending on whether the array is sorted or unsorted:

a. Linear Search

- Suitable for unsorted arrays.
- Traverse the array to find the desired element.
- Example:

```
#include <iostream>
using namespace std;
```

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i; // Found
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    int result = linearSearch(arr, n, key);
    if (result != -1) cout << "Element found at index: " <
< result << endl;
    else cout << "Element not found." << endl;

    return 0;
}
```

Reversing and Rotating Arrays

Reversing an Array

```
#include <iostream>
using namespace std;

void reverseArray(int arr[], int n) {
    int start = 0, end = n - 1;
```

```
    while (start < end) {
        swap(arr[start], arr[end]);
        start++;
        end--;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    reverseArray(arr, n);
    cout << "Reversed array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

Multidimensional Arrays in C++

A **multidimensional array** is an array of arrays, allowing you to store data in more than one dimension. For example, a 2D array can represent a table or matrix, while a 3D array can represent a cube or grid.

How to Declare and Use Multidimensional Arrays ：

1. Declaration

- Multidimensional arrays are declared by specifying multiple sizes in square brackets:

```
data_type array_name[size1][size2][size3];
```

- `size1` represents the first dimension (e.g., rows).
- `size2` represents the second dimension (e.g., columns).
- Additional sizes can be added for more dimensions.

2. Initialization

- Arrays can be initialized during declaration:

```
int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

  - The above is a 2D array with 2 rows and 3 columns.

3. Accessing Elements

- Array elements are accessed using indices:

```
array_name[row_index][column_index];
```

4. Modifying Elements

- You can modify elements by assigning new values to specific indices:

```
array_name[1][2] = 99; // Changes element in row 1, column
2
```

Code Examples

Example 1: Working with a 2D Array

```
#include <iostream>using namespace std;

int main() {
    // Declaration and Initialization of a 2D Array
    int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };

    // Accessing Elements
    cout << "Element at [0][1]: " << arr[0][1] << endl; // Ou
tput: 2

    // Modifying an Element
    arr[1][2] = 99; // Changing the value at [1][2] (row 1, c
olumn 2)
    cout << "Modified element at [1][2]: " << arr[1][2] << en
dl;

    // Printing the Entire Array
    cout << "2D Array:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

What is a Vector in C++?

A **vector** in C++ is a part of the **Standard Template Library (STL)**. It is a **dynamic array** that can grow or shrink in size during runtime. Unlike arrays, vectors do not require the size to be specified during declaration, making them more flexible and easier to use.

Differences Between Arrays and Vectors

Feature	Array	Vector
Size	Fixed-size (declared at compile-time).	Dynamic size (can grow/shrink at runtime).
Memory Management	Memory is statically allocated.	Memory is managed dynamically by the vector class
Operations	Limited in functionality.	Provides many built-in functions like `push_back`, `pop_back`, etc.
Initialization	Needs size during declaration.	No need for size declaration; can initialize directly.
Performance	Faster due to fixed size.	Slightly slower due to dynamic memory management.
Safety	No bounds checking; accessing out-of-bounds elements causes undefined behavior.	Provides optional bounds checking with `at()` method.
Ease of Use	Requires manual memory management for dynamic arrays.	Simplifies dynamic memory management.

When to Use Arrays vs. Vectors

- **Use Arrays** when:
  - The size is fixed and known at compile-time.
  - Performance is critical (e.g., in embedded systems).
- **Use Vectors** when:
  - The size is not known beforehand or can change dynamically.

- You want built-in functionality for easier operations (e.g., insertion, deletion).

How to Declare and Use Vectors

1. Declaration

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v4 = {1, 2, 3, 4}; // Vector initialized with
specific values
    for (int x : v4) {
        cout << x << " "; // Print each element in the vector
    }
    cout << endl;
    return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v1;               // Empty vector of integer
s
    vector<int> v2(5);            // Vector with 5 elements,
all initialized to 0
```

```
    vector<int> v3(5, 10);         // Vector with 5 elements,
all initialized to 10
    vector<int> v4 = {1, 2, 3, 4}; // Vector initialized with
specific values
    cout<< v1.size() << endl;       // Output: 0
    cout << v2.size() << endl;       // Output: 5
    cout << v3.size() << endl;       // Output: 5
    cout << v4.size() << endl;       // Output: 4
    for (int x : v4) {
        cout << x << " "; // Print each element in the vector
    }
    cout << endl;
    return 0;
}
```

4. Traversing a Vector

- **Using a Loop**:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
```

- **Using Range-Based Loop**:

```
cpp
CopyEdit
for (int x : v) {
    cout << x << " ";
}
```

3. Accessing Elements

- **Using Index**:

```
cpp
CopyEdit
cout << v[0];   // Access first element
cout << v.at(1); // Safe access to the second element (che
cks bounds) at index value
```

- **Front and Back**:

```
cout << v.front(); // First element
cout << v.back();  // Last element
```

3. Adding Elements

- Add elements using `push_back()`:

```
vec.push_back(5);  // Adds 5 at the end of the vector
```

2. Adding and Removing Elements

- `push_back(element)`: Adds an element to the end of the vector.
- `pop_back()`: Removes the last element from the vector.

```
cpp
CopyEdit
int main() {
    vector<int> v;

    v.push_back(10); // Add 10
```

```
    v.push_back(20); // Add 20
    v.push_back(30); // Add 30

    cout << "Vector size: " << v.size() << endl; // Prints 3

    v.pop_back(); // Remove last element (30)
    cout << "After pop_back, size: " << v.size() << endl; //
Prints 2

    return 0;
}
```

5. Modifying Elements

- Modify using indexing:

```
vec[1] = 10;  // Changes the second element to 10
```

Code Example: Comparison of Arrays and Vectors

```
#include <iostream>#include <vector>  // Include the vector l
ibraryusing namespace std;

int main() {
    // Array
    int arr[5] = {1, 2, 3, 4, 5};
    cout << "Array elements: ";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
```

```
    cout << endl;

    // Vector
    vector<int> vec = {1, 2, 3, 4, 5};
    cout << "Vector elements: ";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    cout << endl;

    // Adding an element to the vector
    vec.push_back(6);
    cout << "After adding an element, vector elements: ";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    cout << endl;

    // Modifying an element in the vector
    vec[2] = 99;  // Change the 3rd element to 99
    cout << "After modification, vector elements: ";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Advantages of Vectors Over Arrays

1. **Dynamic Sizing:**

- Vectors can automatically resize as elements are added or removed.

2. **Built-In Functions:**
   - Vectors provide a wide range of useful functions like `push_back()`, `pop_back()`, `size()`, `clear()`, and more.
3. **Bounds Checking:**
   - Using `at()` ensures safe access, avoiding segmentation faults for out-of-bounds indices.
4. **Ease of Use:**
   - Vectors simplify memory management, making the code cleaner and more readable.

Static and Dynamic allocation:(runtime increment in memory and also double the size of when doing push and pop operation)

1. Static Allocation

Explanation:

- Memory is allocated **at compile time**.
- Variables are created in the stack or global memory.
- The size of variables or arrays is fixed and cannot change during runtime.

Example Code:

```
cpp
CopyEdit
#include <iostream>using namespace std;

int main() {
    int staticVar = 10;      // Static allocation for an inte
ger
    int staticArray[5] = {1, 2, 3, 4, 5}; // Static allocatio
n for an array

    // Access and modify static variables
    cout << "Static variable: " << staticVar << endl;
    cout << "Static array: ";
    for (int i = 0; i < 5; i++) {
        cout << staticArray[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
sql
CopyEdit
Static variable: 10
Static array: 1 2 3 4 5
```

---

2. Dynamic Allocation

Explanation:

- Memory is allocated **at runtime** using `new` or `malloc` (in C++).
- The size can be determined dynamically (e.g., based on user input).
- Memory is allocated on the heap, and the programmer is responsible for freeing it using `delete` (or `free` in C).
- Offers flexibility, but improper handling can cause memory leaks.

Example Code:

```
cpp
CopyEdit
#include <iostream>using namespace std;

int main() {
    // Dynamic allocation of a single integer
    int* dynamicVar = new int(20);

    // Dynamic allocation of an array
    int size;
    cout << "Enter the size of the dynamic array: ";
    cin >> size;
    int* dynamicArray = new int[size];

    // Assigning values to the array
    for (int i = 0; i < size; i++) {
        dynamicArray[i] = i + 1;
    }

    // Access and print dynamic variables
    cout << "Dynamically allocated variable: " << *dynamicVar
<< endl;
    cout << "Dynamically allocated array: ";
    for (int i = 0; i < size; i++) {
        cout << dynamicArray[i] << " ";
    }
```

```
    cout << endl;

    // Free allocated memory
    delete dynamicVar;
    delete[] dynamicArray;

    return 0;
}
```

Output:

```
sql
CopyEdit
Enter the size of the dynamic array: 5
Dynamically allocated variable: 20
Dynamically allocated array: 1 2 3 4 5
```

Key Differences Between Static and Dynamic Allocation

Aspect	Static Allocation	Dynamic Allocation
Memory Location	Allocated on the stack or global memory.	Allocated on the heap.
Size	Fixed at compile time.	Can be determined at runtime.
Lifetime	Automatically managed (deallocated when out of scope).	Manually managed (must free memory).
Flexibility	Limited (size cannot change).	Very flexible.
Syntax	Direct initialization (e.g., `int x;`).	Use of `new` (e.g., `int* x = new int;`).