Using Array (Vector in C++)

Explanation

An array is a collection of elements of the same data type stored in contiguous memory. In modern C++, arrays are commonly implemented using vector from the STL. A vector is a dynamic array that can grow or shrink in size automatically. It provides built-in functions for easier data management. Unlike static arrays, vectors handle memory allocation internally. Elements are accessed using index positions starting from 0. Vectors are widely used because they are safer and more flexible than traditional arrays.

Advantages

- Dynamic size (can grow/shrink)
- Built-in functions (push_back, size, etc.)
- Safer than raw arrays
- Easy to use and manage

Disadvantages

- Slightly slower than static arrays
- Uses extra memory
- Less control over memory

Syntax

```C++
#include <vector>
using namespace std;

vector<int> arr;          // empty vector
vector<int> arr2 = {1,2,3};  // initialized vector
```

📌 Examples

```cpp
vector<int> numbers = {10, 20, 30};
cout << numbers[0];  // Output: 10
```

```cpp
vector<string> names = {"Ali", "Ahmad"};
cout << names[1];  // Output: Ahmad
```

```cpp
vector<float> marks(3); // size 3
marks[0] = 90.5;
```

Insertion in Array

Explanation

Insertion means adding elements into an array or vector. In vectors, insertion is simple using built-in functions. The most common method is push_back() which adds at the end. You can also insert at a specific position using insert(). Insertion may shift elements if done in the middle. Vectors automatically resize when capacity is exceeded. This makes insertion efficient and flexible compared to static arrays.

Advantages

- Easy insertion with built-in functions
- Automatic resizing
- Flexible positions

Disadvantages

- Slower for middle insertion
- Memory reallocation overhead

Syntax

```cpp
arr.push_back(value);
arr.insert(arr.begin() + index, value);
```

📌 Examples

```cpp
vector<int> arr;
arr.push_back(10);
arr.push_back(20);
```

```cpp
vector<int> arr = {1,2,3};
arr.insert(arr.begin() + 1, 5);  // insert 5 at index 1
```

```cpp
vector<string> v;
v.push_back("Hello");
```

Remove from Array

Explanation

> Removing means deleting elements from an array/vector. In vectors, removal is done using pop_back() or erase(). pop_back() removes the last element. erase() removes element at a specific position. Removing elements shifts remaining elements. Vector size decreases automatically after removal. This simplifies memory management compared to static arrays.

Advantages

- Easy removal operations
- Automatic resizing
- Multiple removal methods

Disadvantages

- Slow for large arrays (shifting)
- Index changes after removal

Syntax

```cpp
arr.pop_back();
arr.erase(arr.begin() + index);
```

📌 Examples

```cpp
vector<int> arr = {10,20,30};
arr.pop_back();  // removes 30
```

```cpp
vector<int> arr = {1,2,3};
arr.erase(arr.begin() + 1);  // removes 2
```

```cpp
vector<string> v = {"A","B","C"};
v.erase(v.begin());
```

Get Size of Array

Explanation

> Size means the number of elements in the array/vector. In vectors, size is obtained using size() function. It returns the current number of elements. Size changes dynamically with insertion/removal. Helps in iteration and bounds checking. Prevents accessing invalid indexes. Important for writing safe and efficient programs.

Advantages

- Easy to get size
- Always accurate
- Useful in loops

Disadvantages

- Slight overhead compared to static arrays

Syntax

```cpp
arr.size();
```

📌 Examples

```cpp
vector<int> arr = {1,2,3};
cout << arr.size();  // Output: 3
```

```cpp
vector<string> v;
v.push_back("Hi");
cout << v.size();  // Output: 1
```

```cpp
vector<int> v(5);
cout << v.size();  // Output: 5
```

Two-Dimensional Array (2D Vector)

Explanation (7 lines)

A 2D array is an array of arrays (matrix form). In C++, it is implemented using vector of vectors. Used to represent tables, matrices, grids. Elements are accessed using row and column index. Structure is like rows and columns. Dynamic resizing is possible in both dimensions. Widely used in algorithms, games, and data representation.

Advantages

- Flexible size
- Easy matrix representation
- Dynamic memory

Disadvantages

- More complex than 1D array

- Slightly slower

Syntax

```cpp
vector<vector<int>> matrix;
vector<vector<int>> mat = {{1,2},{3,4}};
```

📌 Examples

```cpp
vector<vector<int>> mat = {
    {1,2},
    {3,4}
};
```

```cpp
cout << mat[0][1];  // Output: 2
```

```cpp
vector<vector<int>> grid(3, vector<int>(3, 0));
```