Vectors in C++

By: Aaryan Vinod Kumar

A **vector** in C++ is a dynamic array provided by the Standard Template Library (STL). Unlike static arrays, vectors can grow or shrink in size at runtime, which makes them extremely useful when the number of elements isn't known beforehand.

Why Use Vectors?

- **Dynamic Sizing**: Vectors automatically increase or decrease their size when elements are added or removed, avoiding the limitations of fixed-size arrays.
- **Built-in Utility Functions**: Vectors come with many member functions like `.push_back()`, `.pop_back()`, `.size()`, etc., to make managing elements easier.
- **Efficient Memory Management**: Internally, vectors handle memory reallocation and resizing automatically.
- **Random Access**: You can access any element directly using an index, just like in arrays.

Key Terms Explained

- **Dynamic Array**: An array whose size can be changed during runtime.
- **Homogeneous Data**: A data structure is *homogeneous* if it can store only elements of the same data type. For example, a `vector<int>` can only hold integers.
- **Overhead**: Overhead refers to the extra computation or memory usage required to manage something. For vectors, overhead comes from dynamically allocating memory, resizing, copying elements, etc.

Syntax:

```
#include <vector>

using namespace std;


// Declaration

vector<int> v;


// Initialization

vector<int> v1 = {1, 2, 3};

vector<int> v2(5);        // 5 elements, all 0

vector<int> v3(5, -1);    // 5 elements, all -1
```

Common Vector Functions:

Function	Description
`v.size()`	Returns the number of elements in the vector
`v.push_back(x)`	Adds element `x` at the end of the vector
`v.pop_back()`	Removes the last element from the vector
`v.clear()`	Removes all elements, vector becomes empty
`v.empty()`	Returns `true` if the vector is empty, `false` otherwise
`v.front()`	Returns the first element
`v.back()`	Returns the last element
`v.at(index)`	Returns the element at given index, with bounds checking
`v.begin(), v.end()`	Return iterators to the start and end of the vector

Traversing a Vector

Using Index:

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

Range-Based Loop:

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

Using Iterators:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
```

Sample Program: Input and Output

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

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    vector<int> v;

    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        v.push_back(x);
    }
```

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

    return 0;
}
```

Output:

```
Enter number of elements: 5
Enter elements: 1 2 15 -90 3
You entered: 1 2 15 -90 3

=== Code Execution Successful ===
```

Disadvantages of Vectors

- **Slower Insertions in Middle or Beginning**: Vectors are optimized for adding/removing elements at the end. Inserting in the middle or beginning requires shifting elements, which takes more time.

- **Overhead Due to Memory Reallocation**: When a vector runs out of space, it allocates a larger block of memory, copies existing elements, and then continues. This copying is called *reallocation overhead*.

- **Only Homogeneous Data**: A single vector can only hold elements of the same type. If you need to store different types together (e.g., `int` and `string`), you must use a pair or a struct.

Real-World Use Cases

- **Student Score Management** – Storing variable-length test scores.

- **Inventory Systems** – Dynamic lists of items in a game or store.

- **Dynamic Graphs** – Adjacency lists for graphs use vectors to represent edges.

- **To-Do Apps** – Lists where tasks can be added or removed frequently.

Problem Statement

Write a C++ program to rotate a vector to the right by one position.
The last element should move to the front, and all others should shift one step to the right.

Solution to question in the previous lesson:

Question:

Write a C++ program to shift all elements of an array one position to the right. The last element should become the first element. Take the elements from the user.

Program:

```cpp
#include <iostream>

using namespace std;

int main() {

    int n;

    cout << "Enter number of elements: ";

    cin >> n;


    int arr[n];


    cout << "Enter elements: ";
```

```
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int last_element = arr[n - 1];

    // Shift elements to the right
    for (int i = n - 1; i > 0; i--) {
        arr[i] = arr[i - 1];
    }

    arr[0] = last_element;

    cout << "Array after right rotation:\n";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << ' ';
    }

    return 0;
}
```

Output:

```
Enter number of elements: 5
Enter elements: 1 2 3 4 5
Array after right rotation:
5 1 2 3 4

=== Code Execution Successful ===
```