No, `vector<int> adj[]` and `vector<vector<int>> adj` are not the same in C++.

Here are the differences:

1. `vector<int> adj[]`

- This declares an array of `vector<int>` objects, where the size of the array is **fixed** at compile time. The number of elements in the array must be known beforehand.

Example:
cpp
Copy code

```
vector<int> adj[6];  // Array of 6 vectors of integers
```

-
- Here, you have an array of 6 vectors, where each vector can hold integers, and the number of vectors is fixed.

2. `vector<vector<int>> adj`

- This declares a **dynamic 2D vector**, which is a vector of vectors of integers.
- The size of the outer vector (i.e., the number of inner vectors) can be changed at runtime.

Example:
cpp
Copy code

```
vector<vector<int>> adj(6);  // 2D vector with 6 inner vectors
```

-
- Here, the outer `vector` contains 6 inner `vector<int>` elements, and the size of both the outer and inner vectors can change dynamically as needed.

Key Differences:

1. **Static vs Dynamic**:
    - `vector<int> adj[]`: The size of the array is static (fixed at compile time).
    - `vector<vector<int>> adj`: The size of the outer vector can be dynamic (resized at runtime).
2. **Initialization**:
    - `vector<int> adj[]` requires the size of the array to be specified either at the time of declaration or in a different context. For example, you cannot resize the array once declared.

    - `vector<vector<int>> adj` allows resizing both the outer and inner vectors at runtime, providing more flexibility.
3. **Memory Management**:
    - `vector<int> adj[]` is allocated on the stack, and the size of the array is fixed.
    - `vector<vector<int>> adj` is allocated on the heap and dynamically resizable, making it more flexible but with more overhead.

Example for `vector<int> adj[]`:

cpp
Copy code

```
// Array of vectors, size is fixed (say 6)
vector<int> adj[6];

// Adding an edge between nodes
adj[0].push_back(1);
adj[1].push_back(0);
```

Example for `vector<vector<int>> adj`:

cpp
Copy code

```
// Dynamic 2D vector, can resize if needed
vector<vector<int>> adj(6);

// Adding an edge between nodes
adj[0].push_back(1);
adj[1].push_back(0);
```

When to Use:

- **Use `vector<int> adj[]`** if you know the size of the graph (number of nodes) at compile time and it won't change.
- **Use `vector<vector<int>> adj`** if you need a dynamically resizable graph or want the flexibility to change the number of nodes (outer vector) and edges (inner vectors) during runtime.

Let me know if you need further explanation or examples!

Yes, both `vector<int> adj[]` and `vector<vector<int>> adj` serve the same functionality in terms of **storing a 2D array (adjacency list for a graph)**. They both store collections of vectors where each vector represents the list of neighbors (or adjacent nodes) for a particular node in a graph.

However, there are differences in their **flexibility**, **memory allocation**, and how they are declared and used. Let's explore their functionalities and how they achieve the same goal of storing 2D data:

1. Functionality (Storing 2D Data)

`vector<int> adj[]`: This is an array of vectors. Each element in the array is a vector that stores the neighbors of a graph node. It functions like a static 2D array.
Example:
cpp
Copy code

```
vector<int> adj[6];  // Array of 6 vectors (2D-like structure)

adj[0].push_back(1); // Node 0 is connected to Node 1
adj[1].push_back(0); // Node 1 is connected to Node 0
adj[1].push_back(2); // Node 1 is connected to Node 2
```

- Here, each `adj[i]` represents a vector of neighbors for node `i`, achieving the 2D array functionality.

`vector<vector<int>> adj`: This is a dynamic 2D vector. The outer vector contains vectors (each representing neighbors of a node). It achieves the same functionality, but with the flexibility to resize dynamically.
Example:
cpp
Copy code

```
vector<vector<int>> adj(6);  // 2D vector with 6 inner vectors

adj[0].push_back(1); // Node 0 is connected to Node 1
adj[1].push_back(0); // Node 1 is connected to Node 0
adj[1].push_back(2); // Node 1 is connected to Node 2
```

- Again, each `adj[i]` represents the list of neighbors for node `i`, just like in the previous example.

2. Comparison:

Aspect	`vector<int> adj[]`	`vector<vector<int>> adj`
Storage of 2D Data	Yes (array of vectors, each vector stores neighbors)	Yes (2D vector, outer vector stores neighbors)
Fixed Size	Yes, size must be known at compile-time	No, size can be changed at runtime
Resizable	No, the size of the array is fixed	Yes, both the outer and inner vectors can be resized dynamically
Memory Allocation	Stack (size must be known at compile-time)	Heap (dynamic memory, resizable)
Flexibility	Limited (can't resize outer array)	Very flexible (can resize both outer and inner vectors)
Syntax	`vector<int> adj[6];`	`vector<vector<int>> adj(6);`

Key Points:

1. **Same Purpose:** Both are used to represent a 2D structure (e.g., an adjacency list of a graph) and store the same kind of data: each "row" represents the neighbors of a node.
2. **Fixed vs. Dynamic:** `vector<int> adj[]` requires the size to be fixed and known at compile-time, while `vector<vector<int>> adj` is more flexible as it allows the size of the outer vector to change dynamically.
3. **Memory Usage:** `vector<int> adj[]` allocates memory on the stack and requires a fixed size, whereas `vector<vector<int>> adj` allocates memory on the heap, which allows dynamic resizing.

Conclusion:

Both structures can store a 2D array, but **`vector<vector<int>> adj`** is more versatile and commonly used in modern C++ code because it allows dynamic resizing, whereas **`vector<int> adj[]`** is more limited due to its fixed size requirement.

If you need flexibility or are unsure of the graph size at compile-time, `vector<vector<int>>` is a better choice.

Let me know if you have more questions!