Let's break down the provided C++ code for Depth-First Search (DFS) and explain each part with the given example.

Code Breakdown

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

- `#include<iostream>`: Includes the input-output stream library for using `cin` and `cout`.
- `#include<stack>`: Includes the stack library for using the stack data structure.
- `#include<vector>`: Includes the vector library for using the vector data structure.
- `using namespace std;`: Allows using standard library names without prefixing them with `std::`.

```
void dfs(int start, vector<vector<int>>& adj, vector<bool>& visited) {
    stack<int> s;
    s.push(start);
    // visited[start] = true; // Not marked here intentionally for understanding
```

- `void dfs(int start, vector<vector<int>>& adj, vector<bool>& visited)`: Defines the DFS function.
  - `int start`: The starting node for DFS.
  - `vector<vector<int>>& adj`: Adjacency list representation of the graph.
  - `vector<bool>& visited`: Keeps track of visited nodes.
- `stack<int> s;`: Initializes a stack to manage nodes during traversal.
- `s.push(start);`: Pushes the starting node onto the stack.

```
    while (!s.empty()) {
        int node = s.top();
        s.pop();

        if (!visited[node]) {
            visited[node] = true;
            cout << node << "  ";
        }
```

- `while (!s.empty())`: Continues the loop until the stack is empty.
- `int node = s.top();`: Gets the top node from the stack.
- `s.pop();`: Removes the top node from the stack.
- `if (!visited[node])`: Checks if the node hasn't been visited.
  - `visited[node] = true;`: Marks the node as visited.

    - `cout << node << " ";`: Prints the node.

```
        for (int neighbour : adj[node]) {
            if (!visited[neighbour]) {
                s.push(neighbour);
            }
        }
    }
}
```

- `for (int neighbour : adj[node])`: Iterates through all adjacent nodes of the current node.
- `if (!visited[neighbour])`: Checks if the neighbor hasn't been visited.
    - `s.push(neighbour);`: Pushes the unvisited neighbor onto the stack.

```
int main() {
    int n, m;
    cout << "Enter the Number of Nodes:" << endl;
    cin >> n;
    cout << "Enter the Number of Edges" << endl;
    cin >> m;

    vector<vector<int>> adj(n);
    cout << "Enter the edges of (u,v):" << endl;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<bool> visited(n, false);
    dfs(0, adj, visited);

    return 0;
}
```

- `int main()`: The main function where the program starts execution.
- `int n, m;`: Variables for the number of nodes and edges.
- `cout << "Enter the Number of Nodes:" << endl;`: Prompts user for the number of nodes.
- `cin >> n;`: Reads the number of nodes.
- `cout << "Enter the Number of Edges" << endl;`: Prompts user for the number of edges.
- `cin >> m;`: Reads the number of edges.
- `vector<vector<int>> adj(n);`: Initializes the adjacency list for `n` nodes.
- `cout << "Enter the edges of (u,v):" << endl;`: Prompts user to input edges.
- `for (int i = 0; i < m; i++)`: Loop to read `m` edges.
    - `int u, v;`: Variables for each edge.

    - `cin >> u >> v;`: Reads edge (u, v).
    - `adj[u].push_back(v);`: Adds `v` to the adjacency list of `u`.
    - `adj[v].push_back(u);`: Adds `u` to the adjacency list of `v` (since it's an undirected graph).
- `vector<bool> visited(n, false);`: Initializes the visited vector to `false` for `n` nodes.
- `dfs(0, adj, visited);`: Calls the DFS function starting from node 0.

Example Explanation

For the example input:

```
Enter the Number of Nodes:
5
Enter the Number of Edges
5
Enter the edges of (u,v):
0 2
2 1
1 3
4 3
4 2
```

Graph Representation:

```
0 -- 2 -- 1
 \   |
  \  |
   4 -- 3
```

DFS Traversal Steps (starting from node 0):

1. **Start** at node 0, mark it as visited: `Visited = [true, false, false, false, false]`
    - Print 0: `0`
2. **Move** to node 2 (neighbor of 0), mark it as visited: `Visited = [true, false, true, false, false]`
    - Print 2: `0 2`
3. **Move** to node 4 (neighbor of 2), mark it as visited: `Visited = [true, false, true, false, true]`
    - Print 4: `0 2 4`
4. **Move** to node 3 (neighbor of 4), mark it as visited: `Visited = [true, false, true, true, true]`
    - Print 3: `0 2 4 3`
5. **Move** to node 1 (neighbor of 3), mark it as visited: `Visited = [true, true, true, true, true]`
    - Print 1: `0 2 4 3 1`

Output:

```
0 2 4 3 1
```