|  |  |
| --- | --- |
| **Topological sort DFS in C++** | |
| #include <iostream>  #include <vector>  #include <stack>  using namespace std;  class Topo\_dfs {  public:  // Helper function to perform DFS and populate stack  static void dfs(int node, vector<int>& vis, stack<int>& st, vector<vector<int>>& adj) {  vis[node] = 1; // Mark node as visited  // Traverse all adjacent nodes  for (int it : adj[node]) {  if (vis[it] == 0) { // If adjacent node is not visited, perform DFS on it  dfs(it, vis, st, adj);  }  }  st.push(node); // Push current node to stack after visiting all its dependencies  }  // Function to perform topological sorting using DFS  static vector<int> topoSort(int V, vector<vector<int>>& adj) {  vector<int> vis(V, 0); // Initialize visited array  stack<int> st; // Stack to store nodes in topological order  // Perform DFS for each unvisited node  for (int i = 0; i < V; ++i) {  if (vis[i] == 0) {  dfs(i, vis, st, adj);  }  }  vector<int> topo(V);  int index = 0;  // Pop elements from stack to get topological order  while (!st.empty()) {  topo[index++] = st.top();  st.pop();  }  return topo;  }  };  int main() {  int V = 6;  vector<vector<int>> adj(V);  adj[2].push\_back(3);  adj[3].push\_back(1);  adj[4].push\_back(0);  adj[4].push\_back(1);  adj[5].push\_back(0);  adj[5].push\_back(2);  vector<int> ans = Topo\_dfs::topoSort(V, adj);  for (int node : ans) {  cout << node << " ";  }  cout << endl;  return 0;  } | **Revised Dry Run with DFS Call Order**  | **DFS Start** | **Calls** | **Stack Push Order** | | --- | --- | --- | | 0 | No edges → push(0) | 0 | | 1 | No edges → push(1) | 1, 0 | | 2 | DFS(3) → DFS(1) already visited | 3, 2, 1, 0 | | 3 | Already visited |  | | 4 | DFS(0, already visited), DFS(1) | 4, 3, 2, 1, 0 | | 5 | DFS(0, 2) already visited | 5, 4, 3, 2, 1, 0 |  ✅ ****Stack (Top to Bottom)**** 5  4  2  3  1  0 ➡️ ****Final Output**** while (!st.empty()) {  topo[index++] = st.top();  st.pop();  } 🟩 Output: 5 4 2 3 1 0 🧠 Why This Is Valid: Topological sort can have **multiple valid orders** as long as:   * For every edge u → v, u appears **before** v.   And in this case:   * 5 is before 2, 0 * 2 is before 3 * 3 is before 1 * 4 is before 0, 1   ✅ All conditions are satisfied. |
| **Output:-**  5 4 2 3 1 0 | |