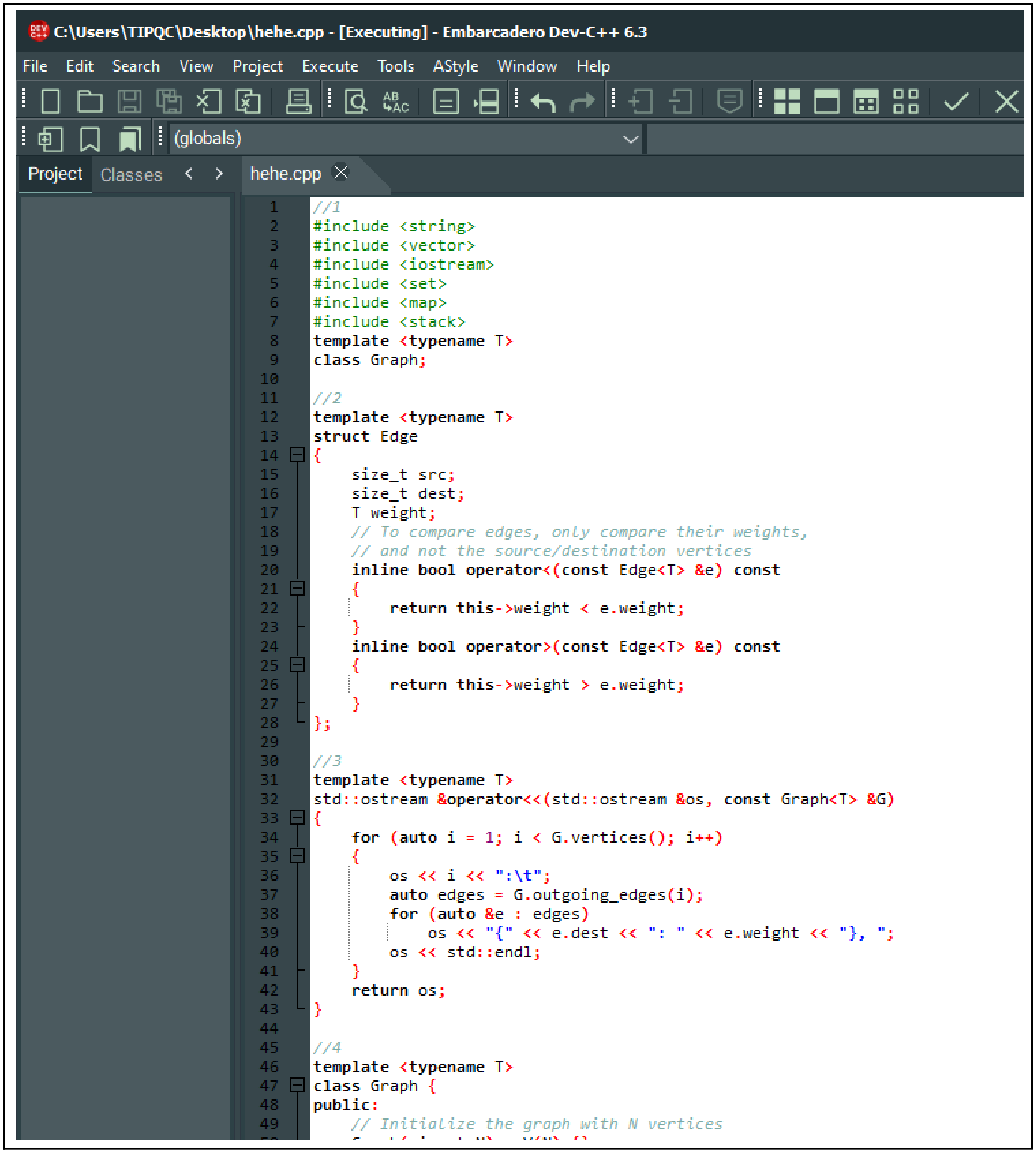
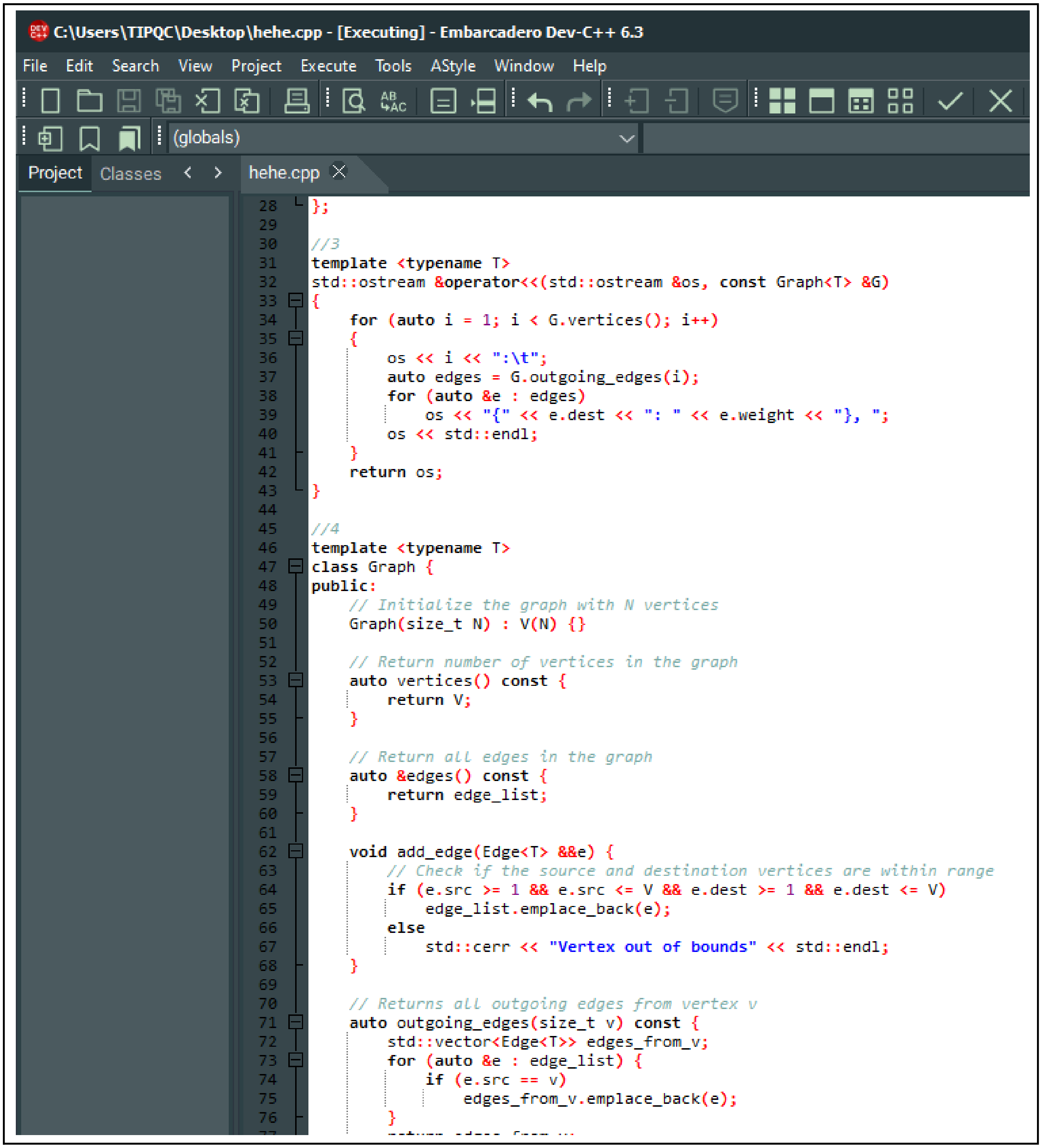
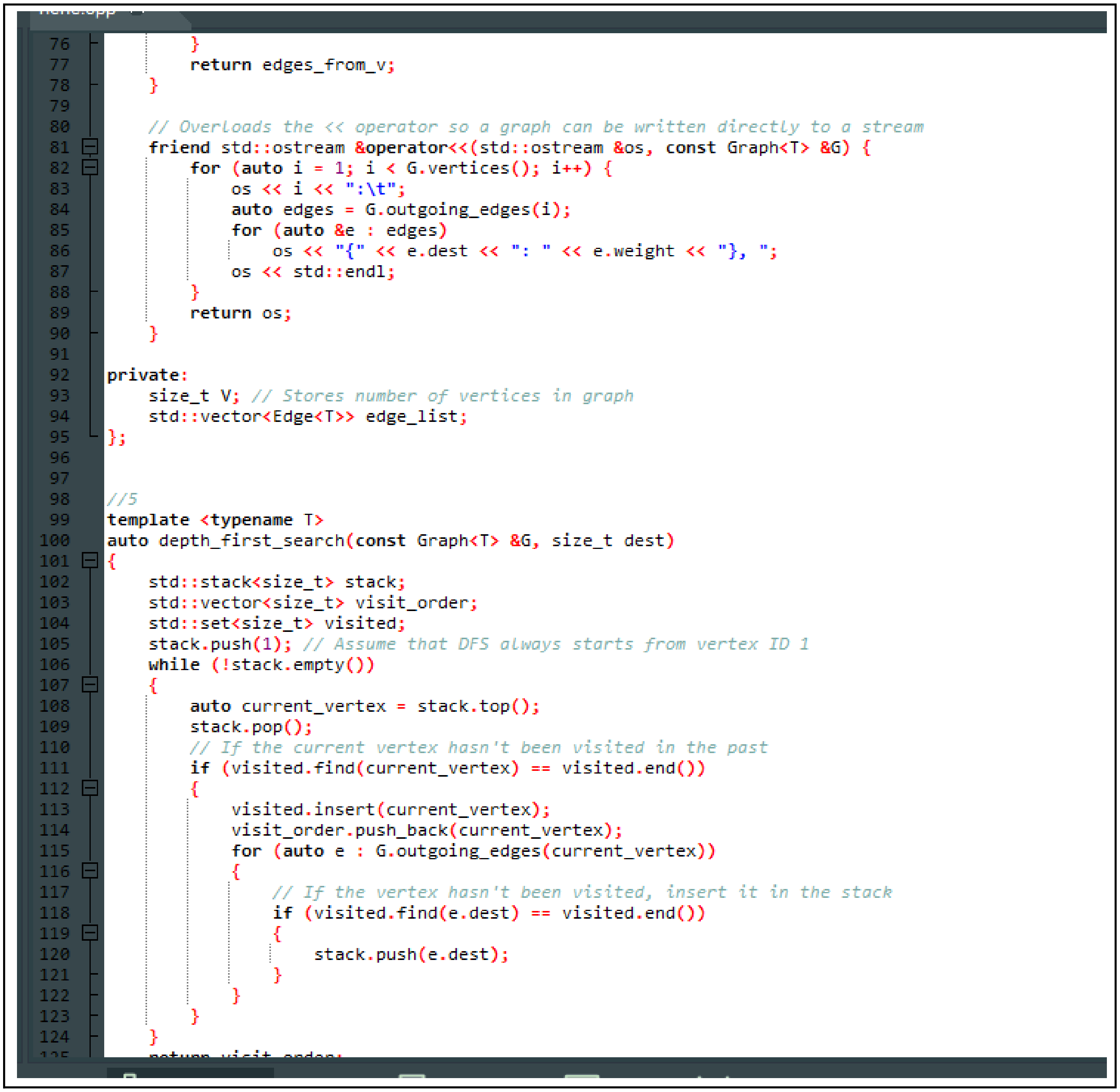
Activity No. 10.2	
Hands-on Activity 10.2 Implementing DFS and Graph Applications	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 11/27/2024**
Section: CPE21S1	**Date Submitted:11/27/2024**
Name(s): Bautista, Jhon Hendricks **Anduque, Kurt Gabriel** **Bonifacio, Redj Guillian** **Magboo, Matt Clemence** **Pateña, Chrisitan Dale**	**Instructor: Mrs. Maria Rizette Sayo**
A. Output(s) and Observation(s)	
Code:	


C:\Users\TIPQC\Desktop\hehe.cpp - [Executing] - Embarcadero Dev-C++ 6.3
File Edit Search View Project Execute Tools AStyle Window Help
(globals)
Project Classes
hehe.cpp

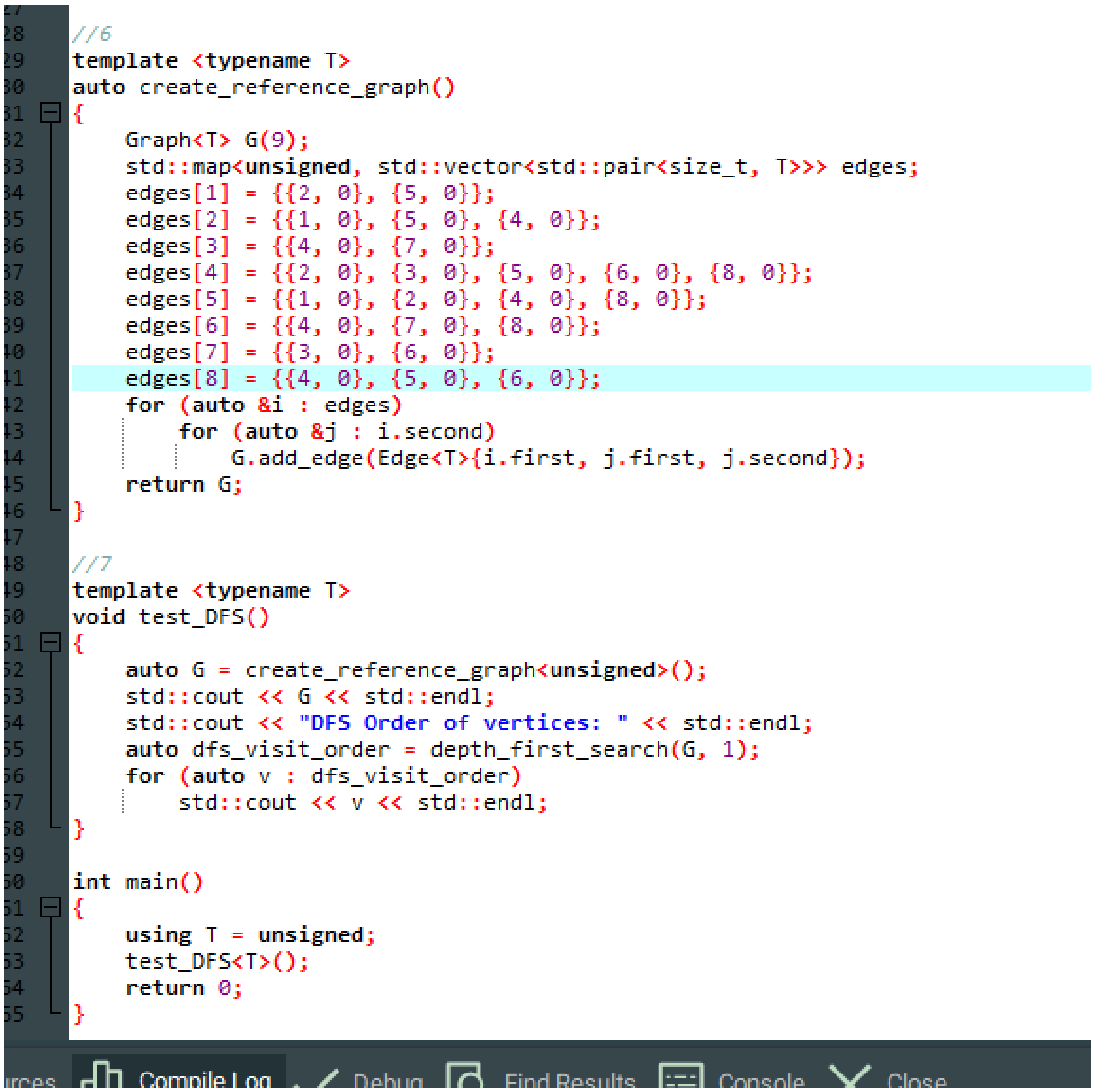
```
//1
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <stack>
template <typename T>
class Graph;

//2
template <typename T>
struct Edge
{
    size_t src;
    size_t dest;
    T weight;
    // To compare edges, only compare their weights,
    // and not the source/destination vertices
    inline bool operator<(const Edge<T> &e) const
    {
        return this->weight < e.weight;
    }
    inline bool operator>(const Edge<T> &e) const
    {
        return this->weight > e.weight;
    }
};

//3
template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i < G.vertices(); i++)
    {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
            os << "{" << e.dest << ": " << e.weight << "}, ";
        os << std::endl;
    }
    return os;
}

//4
template <typename T>
class Graph {
public:
    // Initialize the graph with N vertices
```


C:\Users\TIPQC\Desktop\hehe.cpp - [Executing] - Embarcadero Dev-C++ 6.3

```cpp
};

//3
template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i < G.vertices(); i++)
    {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
            os << "{" << e.dest << ": " << e.weight << "}, ";
        os << std::endl;
    }
    return os;
}

//4
template <typename T>
class Graph {
public:
    // Initialize the graph with N vertices
    Graph(size_t N) : V(N) {}

    // Return number of vertices in the graph
    auto vertices() const {
        return V;
    }

    // Return all edges in the graph
    auto &edges() const {
        return edge_list;
    }

    void add_edge(Edge<T> &&e) {
        // Check if the source and destination vertices are within range
        if (e.src >= 1 && e.src <= V && e.dest >= 1 && e.dest <= V)
            edge_list.emplace_back(e);
        else
            std::cerr << "Vertex out of bounds" << std::endl;
    }

    // Returns all outgoing edges from vertex v
    auto outgoing_edges(size_t v) const {
        std::vector<Edge<T>> edges_from_v;
        for (auto &e : edge_list) {
            if (e.src == v)
                edges_from_v.emplace_back(e);
        }
```

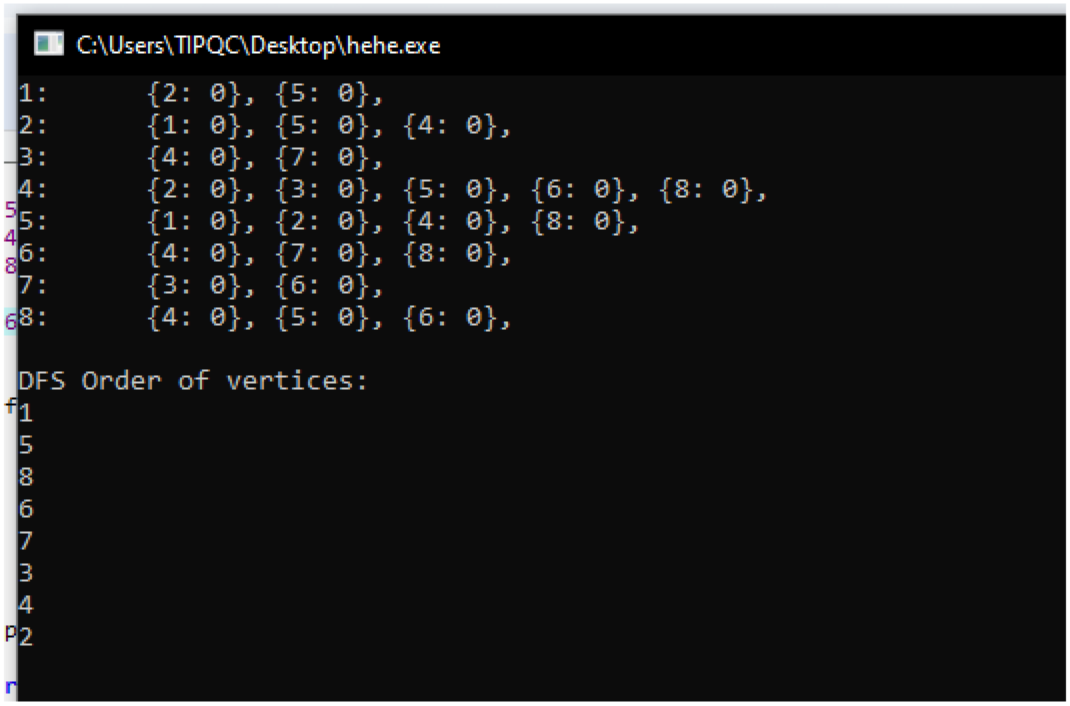


```cpp
        }
        return edges_from_v;
    }

    // Overloads the << operator so a graph can be written directly to a stream
    friend std::ostream &operator<<(std::ostream &os, const Graph<T> &G) {
        for (auto i = 1; i < G.vertices(); i++) {
            os << i << ":\t";
            auto edges = G.outgoing_edges(i);
            for (auto &e : edges)
                os << "{" << e.dest << ": " << e.weight << "}, ";
            os << std::endl;
        }
        return os;
    }

private:
    size_t V; // Stores number of vertices in graph
    std::vector<Edge<T>> edge_list;
};


//5
template <typename T>
auto depth_first_search(const Graph<T> &G, size_t dest)
{
    std::stack<size_t> stack;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;
    stack.push(1); // Assume that DFS always starts from vertex ID 1
    while (!stack.empty())
    {
        auto current_vertex = stack.top();
        stack.pop();
        // If the current vertex hasn't been visited in the past
        if (visited.find(current_vertex) == visited.end())
        {
            visited.insert(current_vertex);
            visit_order.push_back(current_vertex);
            for (auto e : G.outgoing_edges(current_vertex))
            {
                // If the vertex hasn't been visited, insert it in the stack
                if (visited.find(e.dest) == visited.end())
                {
                    stack.push(e.dest);
                }
            }
        }
    }
    return visit_order;
```



```
//6
template <typename T>
auto create_reference_graph()
{
    Graph<T> G(9);
    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
    edges[1] = {{2, 0}, {5, 0}};
    edges[2] = {{1, 0}, {5, 0}, {4, 0}};
    edges[3] = {{4, 0}, {7, 0}};
    edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 0}};
    edges[5] = {{1, 0}, {2, 0}, {4, 0}, {8, 0}};
    edges[6] = {{4, 0}, {7, 0}, {8, 0}};
    edges[7] = {{3, 0}, {6, 0}};
    edges[8] = {{4, 0}, {5, 0}, {6, 0}};
    for (auto &i : edges)
        for (auto &j : i.second)
            G.add_edge(Edge<T>{i.first, j.first, j.second});
    return G;
}

//7
template <typename T>
void test_DFS()
{
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;
    std::cout << "DFS Order of vertices: " << std::endl;
    auto dfs_visit_order = depth_first_search(G, 1);
    for (auto v : dfs_visit_order)
        std::cout << v << std::endl;
}

int main()
{
    using T = unsigned;
    test_DFS<T>();
    return 0;
}
```

Screenshot of Output:

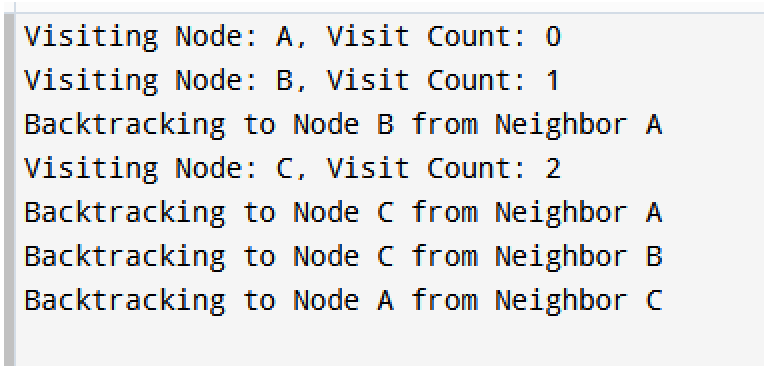


```
C:\Users\TIPQC\Desktop\hehe.exe
1:      {2: 0}, {5: 0},
2:      {1: 0}, {5: 0}, {4: 0},
3:      {4: 0}, {7: 0},
4:      {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0},
5:      {1: 0}, {2: 0}, {4: 0}, {8: 0},
6:      {4: 0}, {7: 0}, {8: 0},
7:      {3: 0}, {6: 0},
8:      {4: 0}, {5: 0}, {6: 0},

DFS Order of vertices:
1
5
8
6
7
3
4
2
```

B. Answers to Supplementary Activity

1. The best way for the person to traverse the path in the map is a depth first search. This is because DFS is effective for exploring all vertices in a connected component of a graph. One factor that contributes to this is it allows backtracking after the traversal of a vertex then its neighbors before returning to explore another vertex. Lastly is it fully explores one branch before moving to another, ensuring all vertices are visited.

2. In the DFS (Depth-First Search) of a graph, within the traversal process can be identified as unique in certain situations if the graph structure and the starting point plays a key role. Meaning that there is no ambiguity on how to visit the nodes, which means that DFS traversal can be considered unique in a tree-like structure.

3. 

```
Visiting Node: A, Visit Count: 0
Visiting Node: B, Visit Count: 1
Backtracking to Node B from Neighbor A
Visiting Node: C, Visit Count: 2
Backtracking to Node C from Neighbor A
Backtracking to Node C from Neighbor B
Backtracking to Node A from Neighbor C
```

The maximum number of visits of vertex in DFS depends on the structure of the graph, especially if the graph contains a cycle

4. DFS can be applied on finding the connection of components, like in an undirected graph DFS can be used to find the connection between components, which means that it will explore all unvisited nodes and reachable nodes after that it will mark those nodes that got visited. This can be useful in finding connection clusters in social networks or identifying isolated groups in network topologies

5. In trees, the comparable algorithm of DFS is a Preorder traversal. This is because Preorder visits the root node first, then recursively visits the left subtree and the right subtree.

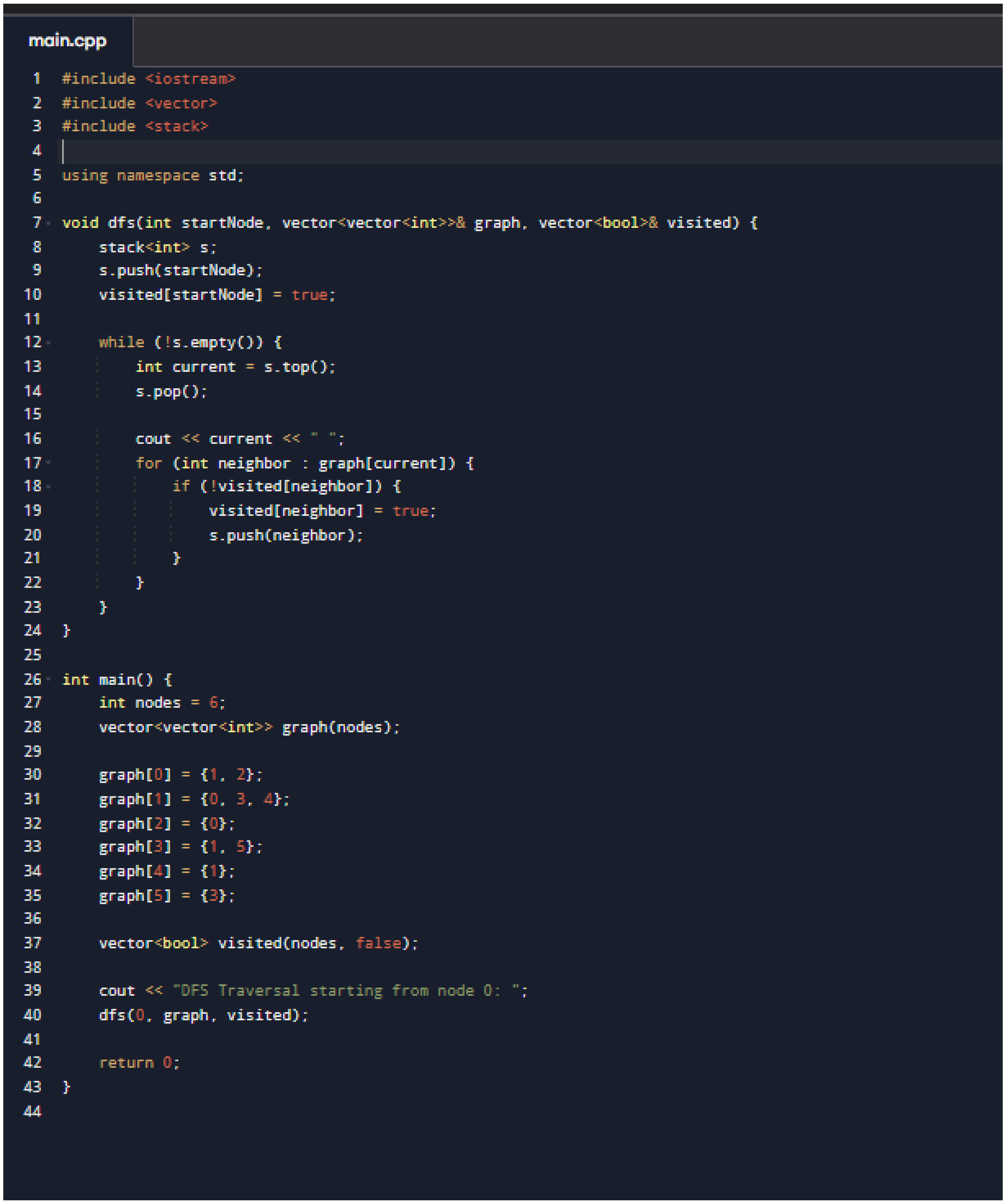
pseudo code for DFS

```
1. Start with the parent node.
2. Create a list or stack to keep track of the nodes you still need to explore.
3. Mark the starting node as visited
4. Take the most recently added node
5. Print it
6. Look for its other neighbors that are not visited
```

pseudo code for Preorder Traversal

```
1. Start with the parent node.
2. Create a list or stack to keep track of nodes you still need to visit.
3. Mark the starting node as visited.
4. For nodes left in the stack: Take the most recently added node from the stack
5. Print the node
6.Add all of its unvisited neighbors to the stack
7. Traverse the leftover nodes
```

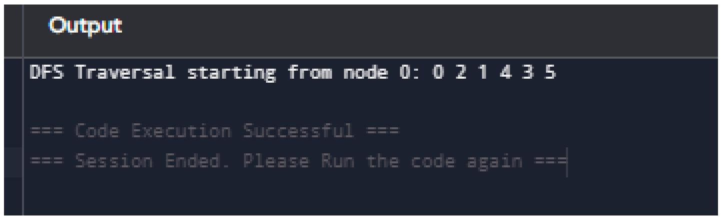
Code for DFS



main.cpp

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

void dfs(int startNode, vector<vector<int>>& graph, vector<bool>& visited) {
    stack<int> s;
    s.push(startNode);
    visited[startNode] = true;

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

        cout << current << " ";
        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                s.push(neighbor);
            }
        }
    }
}

int main() {
    int nodes = 6;
    vector<vector<int>> graph(nodes);

    graph[0] = {1, 2};
    graph[1] = {0, 3, 4};
    graph[2] = {0};
    graph[3] = {1, 5};
    graph[4] = {1};
    graph[5] = {3};

    vector<bool> visited(nodes, false);

    cout << "DFS Traversal starting from node 0: ";
    dfs(0, graph, visited);

    return 0;
}
```

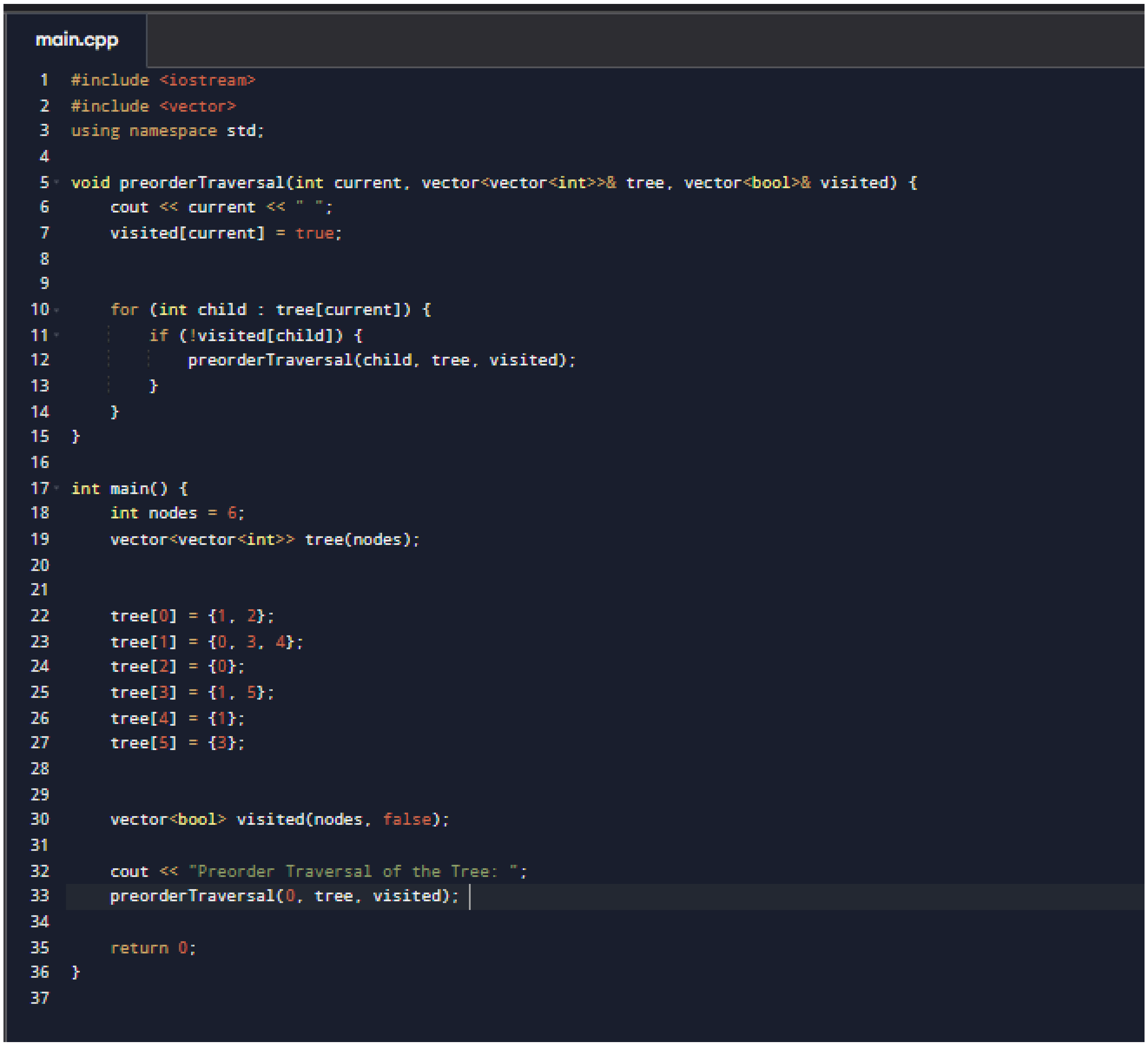
\



Output

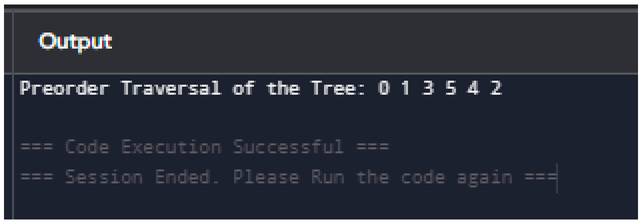
```
DFS Traversal starting from node 0: 0 2 1 4 3 5

=== Code Execution Successful ===
=== Session Ended. Please Run the code again ===
```

Code of Preorder Traversal



main.cpp

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

void preorderTraversal(int current, vector<vector<int>>& tree, vector<bool>& visited) {
    cout << current << " ";
    visited[current] = true;


    for (int child : tree[current]) {
        if (!visited[child]) {
            preorderTraversal(child, tree, visited);
        }
    }
}

int main() {
    int nodes = 6;
    vector<vector<int>> tree(nodes);


    tree[0] = {1, 2};
    tree[1] = {0, 3, 4};
    tree[2] = {0};
    tree[3] = {1, 5};
    tree[4] = {1};
    tree[5] = {3};


    vector<bool> visited(nodes, false);

    cout << "Preorder Traversal of the Tree: ";
    preorderTraversal(0, tree, visited);

    return 0;
}
```



Output

```
Preorder Traversal of the Tree: 0 1 3 5 4 2

=== Code Execution Successful ===
=== Session Ended. Please Run the code again ===
```

C. Conclusion & Lessons Learned
In this hands-on activity focused on implementing Depth-First Search (DFS) and exploring different graph applications, we gained a better understanding of graph traversal techniques and how they can be used in real-life situations. By implementing DFS, we learned how the algorithm explores all the vertices in a connected part of a graph, showing how it backtracks to fully explore one branch before moving on to the next. We learn the basic structure of graphs when using DFS. Also we are now knowledgeable about the syntax of graphs in C++. We also saw how useful DFS can be in real-world applications. Lastly is the comparison between DFS and Preorder Traversal for trees which helped us see how similar algorithms can be used in different situations to achieve similar goals.
D. Assessment Rubric
E. External References