Name: Akash kafle (26), Nabin kafle (27)

Graph

- Graphs are a fundamental data structure in computer science used to represent relationships between objects. They consist of vertices (also called nodes) and edges that connect these vertices.

  - **Key components of a graph:**
    - **Vertices:** The objects or entities in the graph.
    - **Edges:** Connections between vertices.

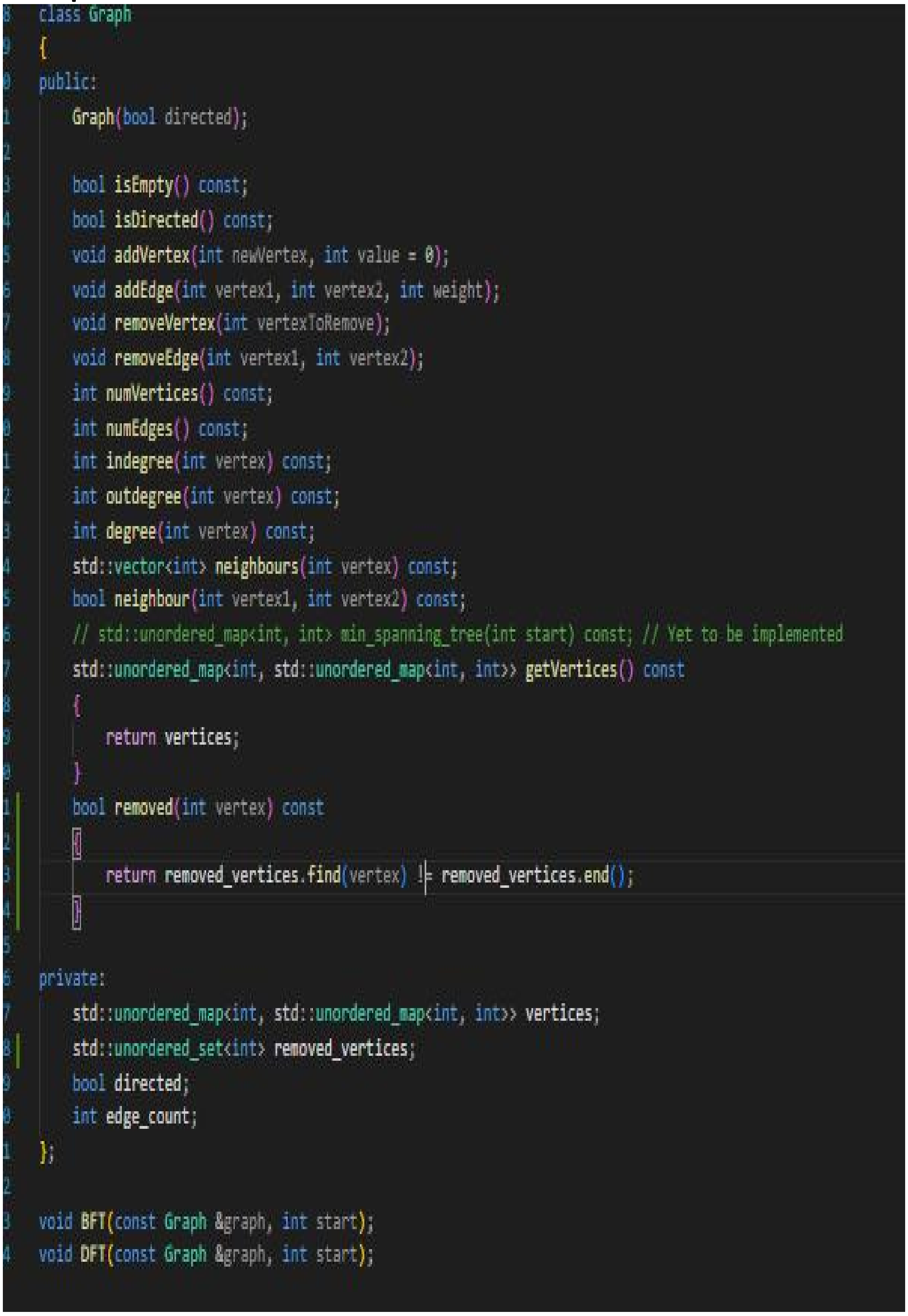
  - **Types of graphs:**
    - **Directed Graph (Digraph)**: Edges have a direction.
    - **Undirected Graph:** Edges have no direction.
    - **Weighted Graph:** Edges have associated weights or costs.

  - **Common representations:**
    - **Adjacency Matrix:** A 2D array where matrix[i][j] represents an edge from vertex i to j.
    - **Adjacency List:** An array of lists, where each list contains the neighbors of a vertex.

- **Basic operations**:
  - Adding vertices and edges
  - Removing vertices and edges
  - Checking if an edge exists between two vertices
  - Finding neighbors of a vertex

- **Common algorithms:**
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
  - Dijkstra's Algorithm (shortest path)
  - Kruskal's and Prim's Algorithms (minimum spanning tree)

Graph.h

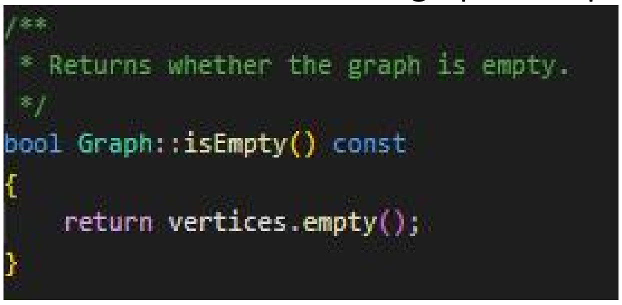


```cpp
class Graph
{
public:
    Graph(bool directed);

    bool isEmpty() const;
    bool isDirected() const;
    void addVertex(int newVertex, int value = 0);
    void addEdge(int vertex1, int vertex2, int weight);
    void removeVertex(int vertexToRemove);
    void removeEdge(int vertex1, int vertex2);
    int numVertices() const;
    int numEdges() const;
    int indegree(int vertex) const;
    int outdegree(int vertex) const;
    int degree(int vertex) const;
    std::vector<int> neighbours(int vertex) const;
    bool neighbour(int vertex1, int vertex2) const;
    // std::unordered_map<int, int> min_spanning_tree(int start) const; // Yet to be implemented
    std::unordered_map<int, std::unordered_map<int, int>> getVertices() const
    {
        return vertices;
    }
    bool removed(int vertex) const
    {
        return removed_vertices.find(vertex) != removed_vertices.end();
    }

private:
    std::unordered_map<int, std::unordered_map<int, int>> vertices;
    std::unordered_set<int> removed_vertices;
    bool directed;
    int edge_count;
};

void BFT(const Graph &graph, int start);
void DFT(const Graph &graph, int start);
```

Graph.cpp

(a) isEmpty():

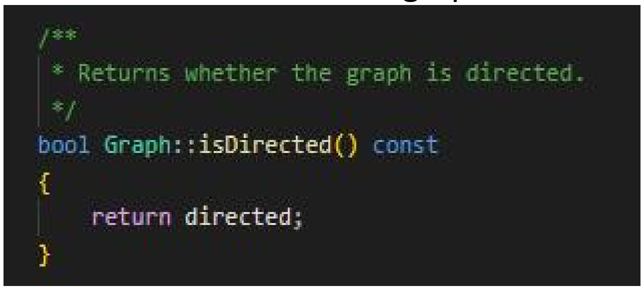
- Returns true if the graph is empty, and false otherwise



```
/**
 * Returns whether the graph is empty.
 */
bool Graph::isEmpty() const
{
    return vertices.empty();
}
```

(b) isDirected():

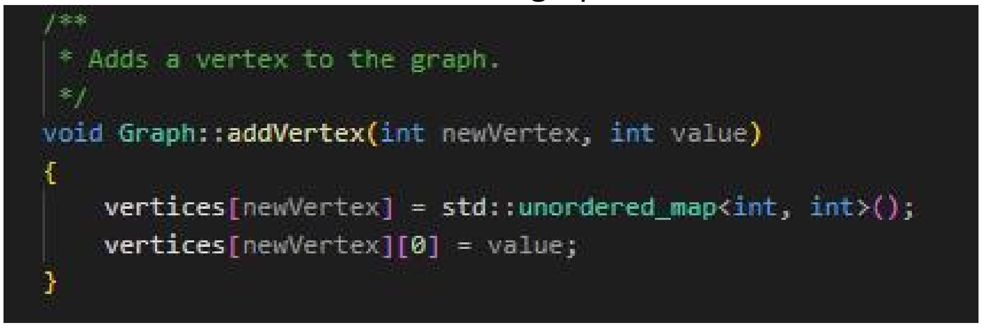
- Returns true if the graph is directed, and false otherwise



```
/**
 * Returns whether the graph is directed.
 */
bool Graph::isDirected() const
{
    return directed;
}
```

(c) addVertex(newVertex):

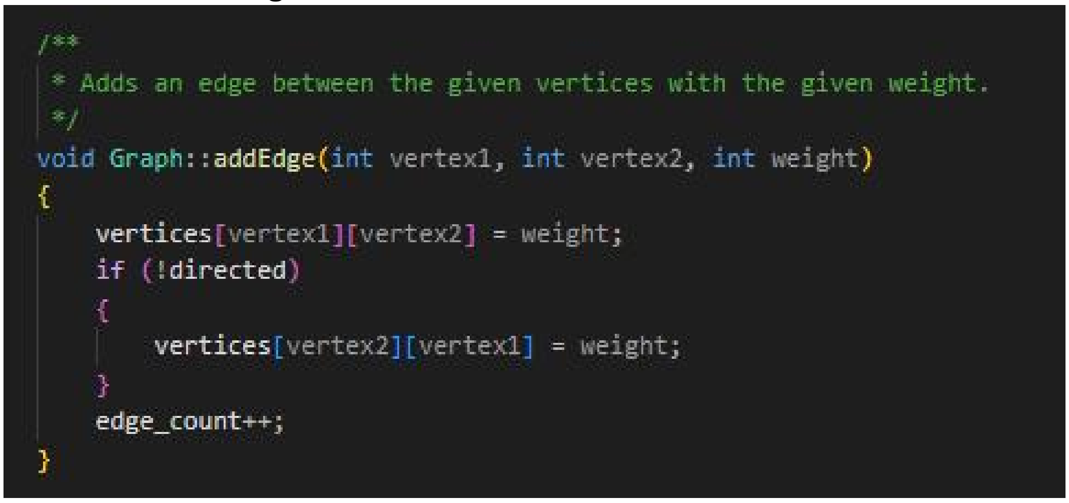
- Inserts a new vertex to the graph



```
/**
 * Adds a vertex to the graph.
 */
void Graph::addVertex(int newVertex, int value)
{
    vertices[newVertex] = std::unordered_map<int, int>();
    vertices[newVertex][0] = value;
}
```

(d) addEdge(vertex1, vertex2):

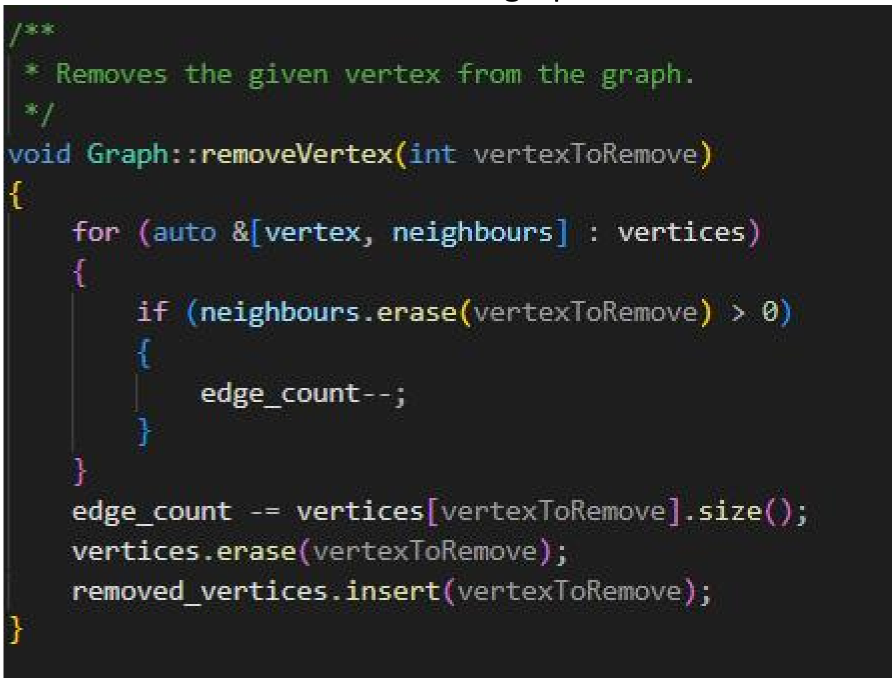
- Adds an edge from vertex1 to vertex2



```
/**
 * Adds an edge between the given vertices with the given weight.
 */
void Graph::addEdge(int vertex1, int vertex2, int weight)
{
    vertices[vertex1][vertex2] = weight;
    if (!directed)
    {
        vertices[vertex2][vertex1] = weight;
    }
    edge_count++;
}
```

(e) removeVertex(vertexToRemove):

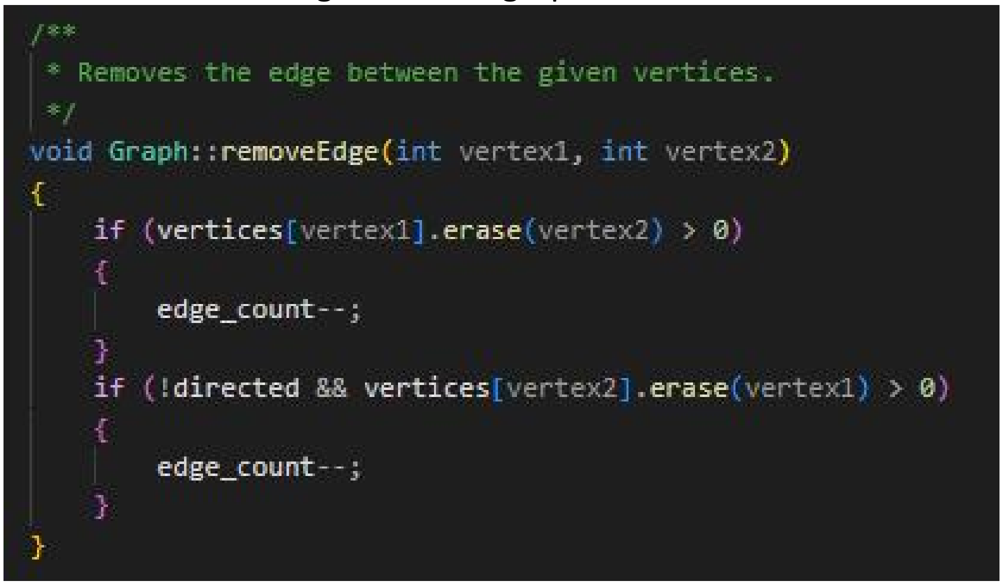
- Remove a vertex from the graph



```cpp
/**
 * Removes the given vertex from the graph.
 */
void Graph::removeVertex(int vertexToRemove)
{
    for (auto &[vertex, neighbours] : vertices)
    {
        if (neighbours.erase(vertexToRemove) > 0)
        {
            edge_count--;
        }
    }
    edge_count -= vertices[vertexToRemove].size();
    vertices.erase(vertexToRemove);
    removed_vertices.insert(vertexToRemove);
}
```

(f) removeEdge(vertex1, vertex2):

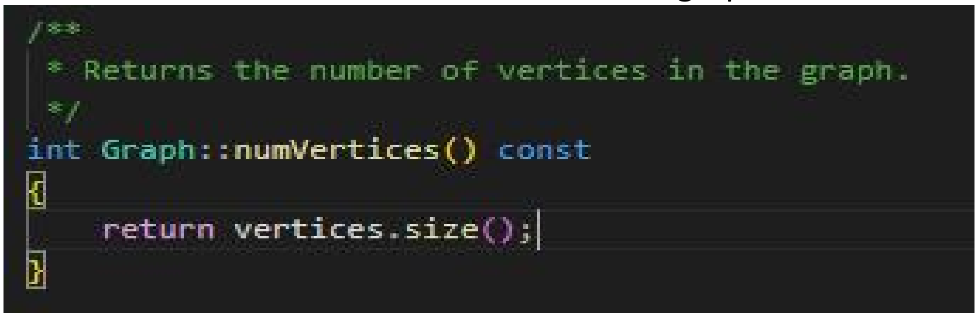
- Remove an edge from the graph



```cpp
/**
 * Removes the edge between the given vertices.
 */
void Graph::removeEdge(int vertex1, int vertex2)
{
    if (vertices[vertex1].erase(vertex2) > 0)
    {
        edge_count--;
    }
    if (!directed && vertices[vertex2].erase(vertex1) > 0)
    {
        edge_count--;
    }
}
```

(g) numVertices():

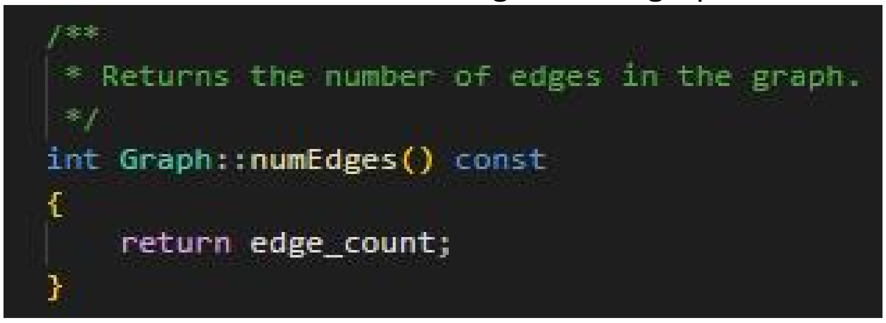
- Returns the number of vertices in the graph



```cpp
/**
 * Returns the number of vertices in the graph.
 */
int Graph::numVertices() const
{
    return vertices.size();
}
```

(h) numEdges():

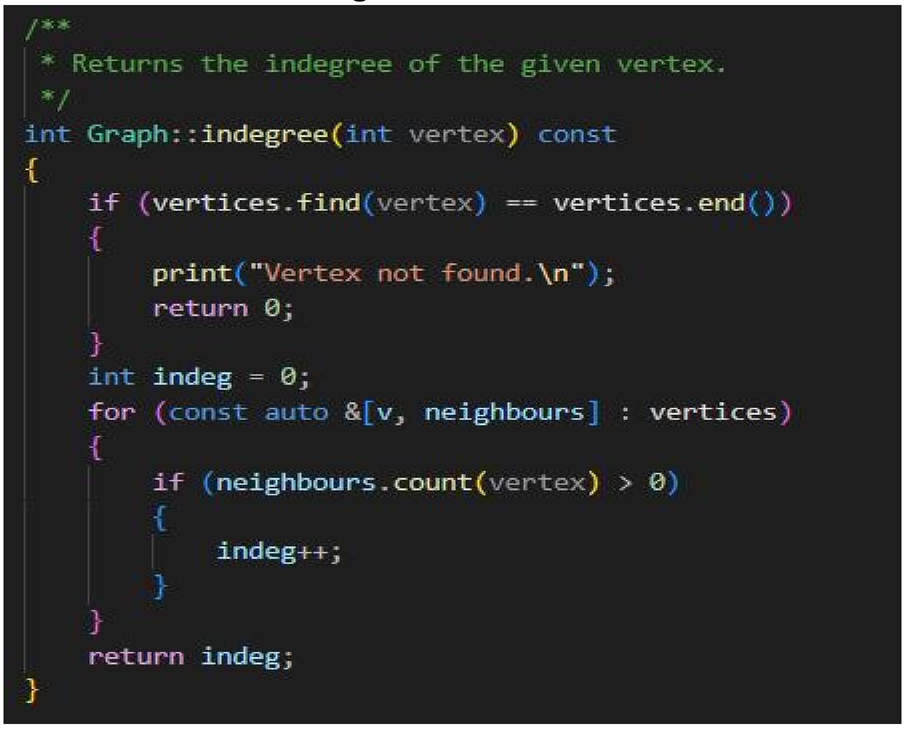
- Returns the number of edges in the graph



```
/**
 * Returns the number of edges in the graph.
 */
int Graph::numEdges() const
{
    return edge_count;
}
```

(i) indegree(vertex):

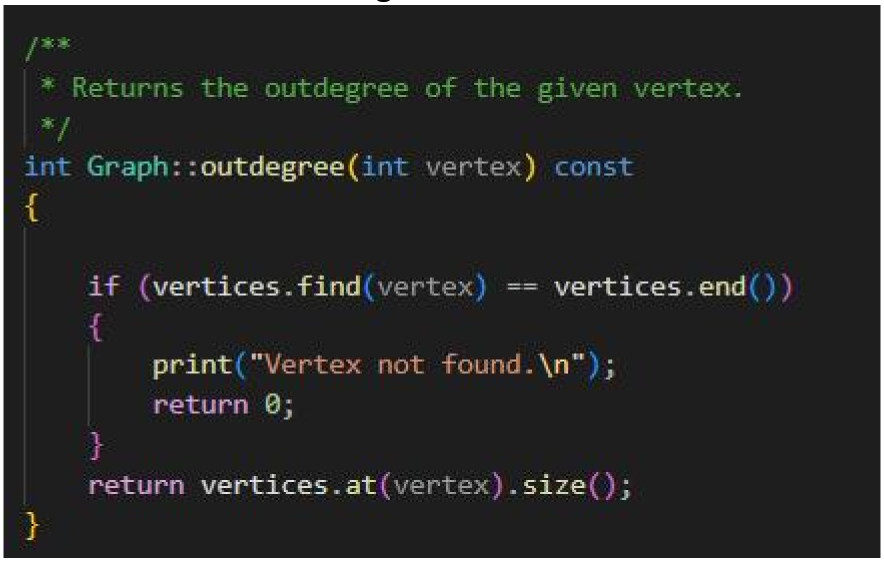
- Returns the indegree of a vertex



```
/**
 * Returns the indegree of the given vertex.
 */
int Graph::indegree(int vertex) const
{
    if (vertices.find(vertex) == vertices.end())
    {
        print("Vertex not found.\n");
        return 0;
    }
    int indeg = 0;
    for (const auto &[v, neighbours] : vertices)
    {
        if (neighbours.count(vertex) > 0)
        {
            indeg++;
        }
    }
    return indeg;
}
```

(j) outdegree(vertex):

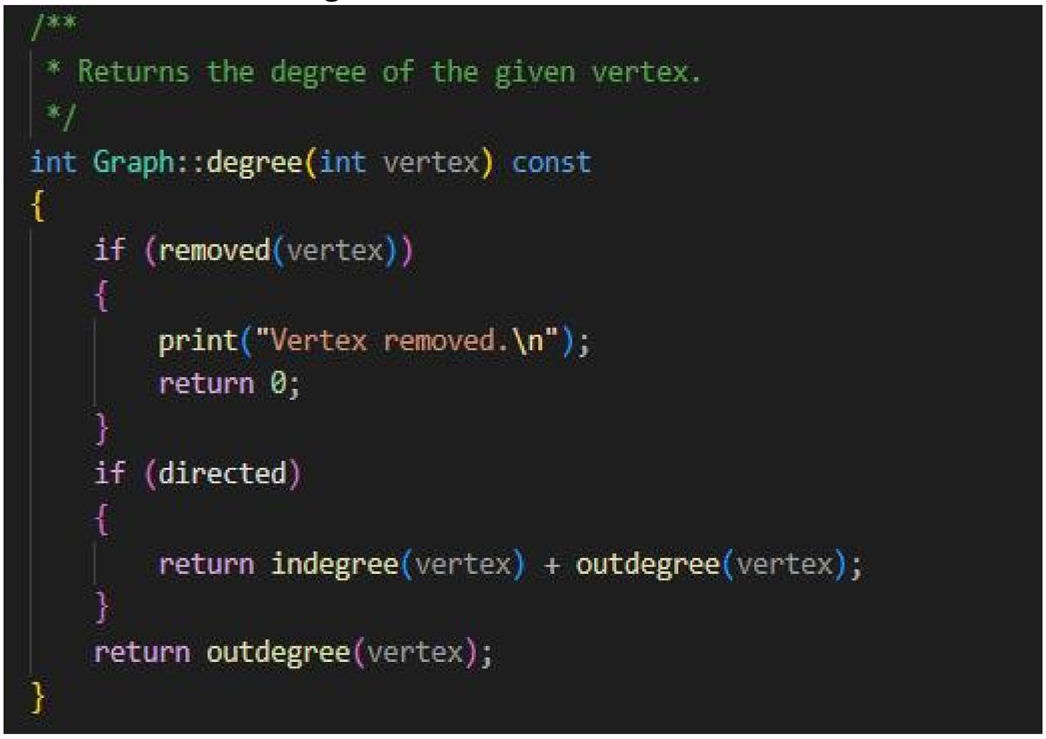
- Returns the outdegree of a vertex



```
/**
 * Returns the outdegree of the given vertex.
 */
int Graph::outdegree(int vertex) const
{

    if (vertices.find(vertex) == vertices.end())
    {
        print("Vertex not found.\n");
        return 0;
    }
    return vertices.at(vertex).size();
}
```

(k) degree(vertex):

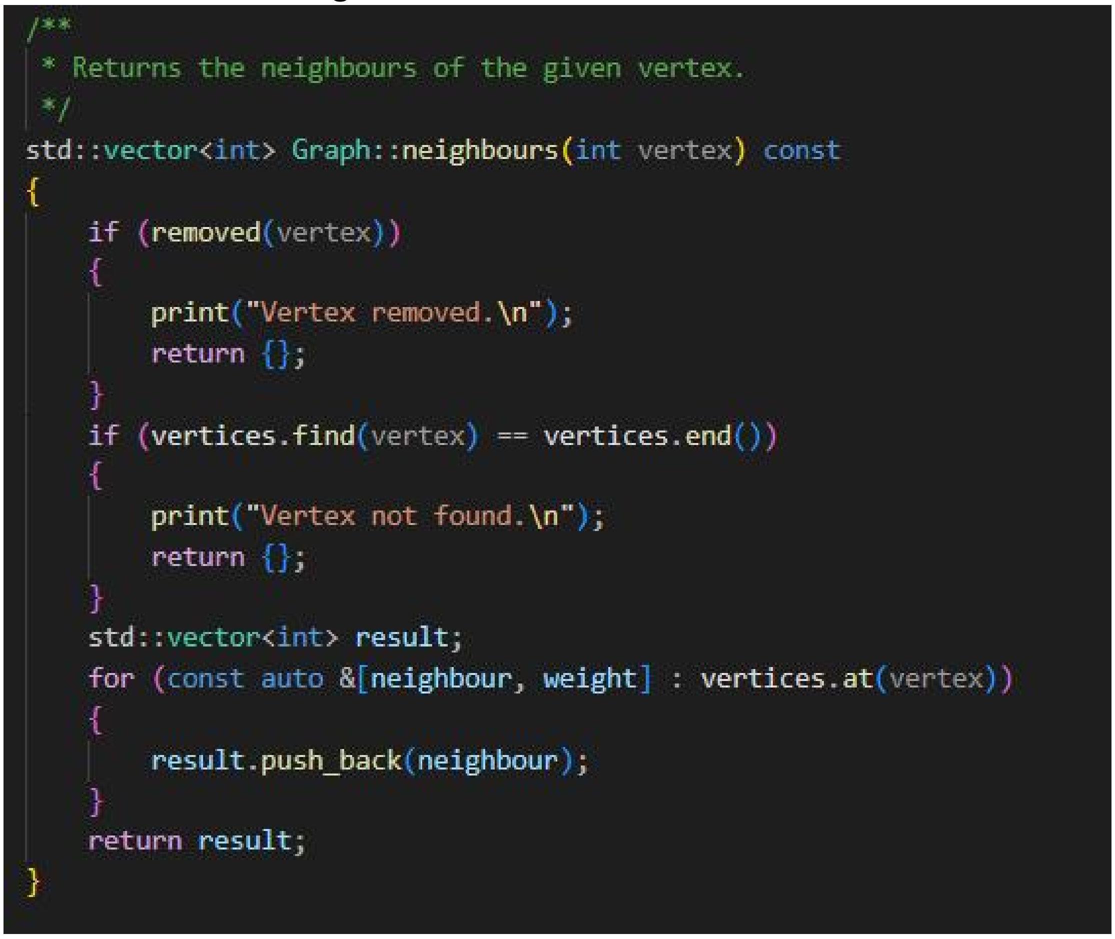
- Returns the degree of a vertex



```cpp
/**
 * Returns the degree of the given vertex.
 */
int Graph::degree(int vertex) const
{
    if (removed(vertex))
    {
        print("Vertex removed.\n");
        return 0;
    }
    if (directed)
    {
        return indegree(vertex) + outdegree(vertex);
    }
    return outdegree(vertex);
}
```

(l) neighbours(vertex):

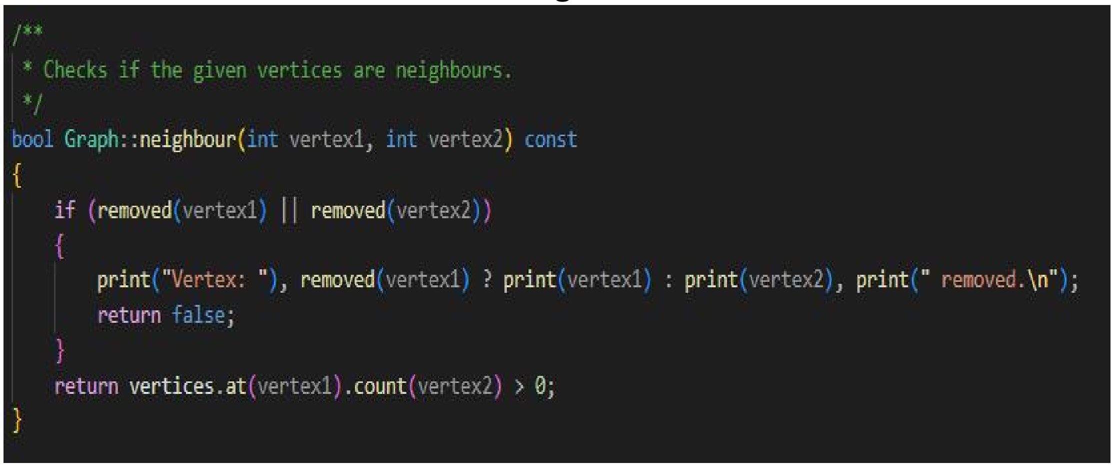
- Returns the neighbours of a vertex



```cpp
/**
 * Returns the neighbours of the given vertex.
 */
std::vector<int> Graph::neighbours(int vertex) const
{
    if (removed(vertex))
    {
        print("Vertex removed.\n");
        return {};
    }
    if (vertices.find(vertex) == vertices.end())
    {
        print("Vertex not found.\n");
        return {};
    }
    std::vector<int> result;
    for (const auto &[neighbour, weight] : vertices.at(vertex))
    {
        result.push_back(neighbour);
    }
    return result;
}
```

(m) neighbour(vertex1, vertex2):

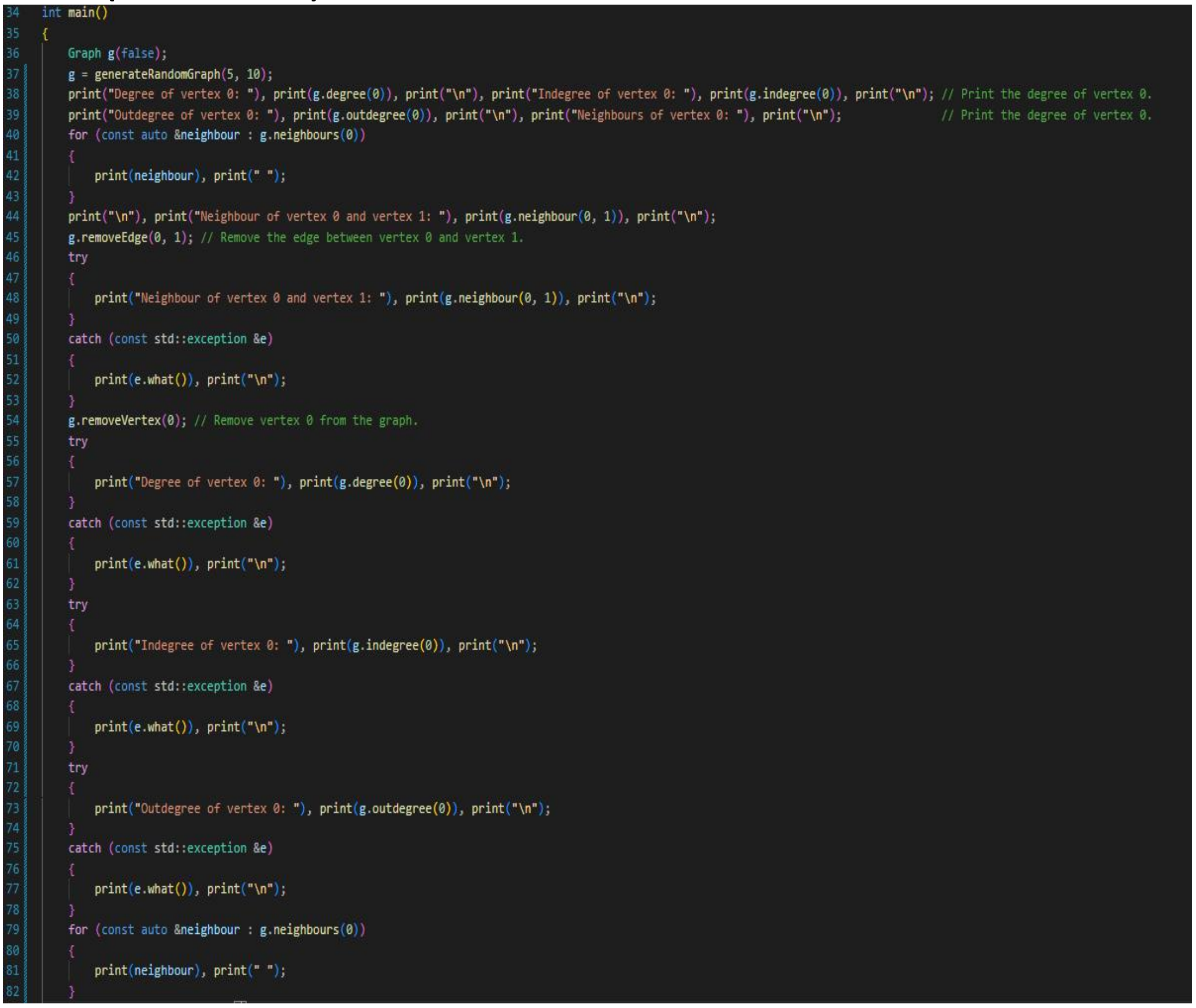
- Returns true if vertex2 is a neighbour of vertex1.



```cpp
/**
 * Checks if the given vertices are neighbours.
 */
bool Graph::neighbour(int vertex1, int vertex2) const
{
    if (removed(vertex1) || removed(vertex2))
    {
        print("Vertex: "), removed(vertex1) ? print(vertex1) : print(vertex2), print(" removed.\n");
        return false;
    }
    return vertices.at(vertex1).count(vertex2) > 0;
}
```

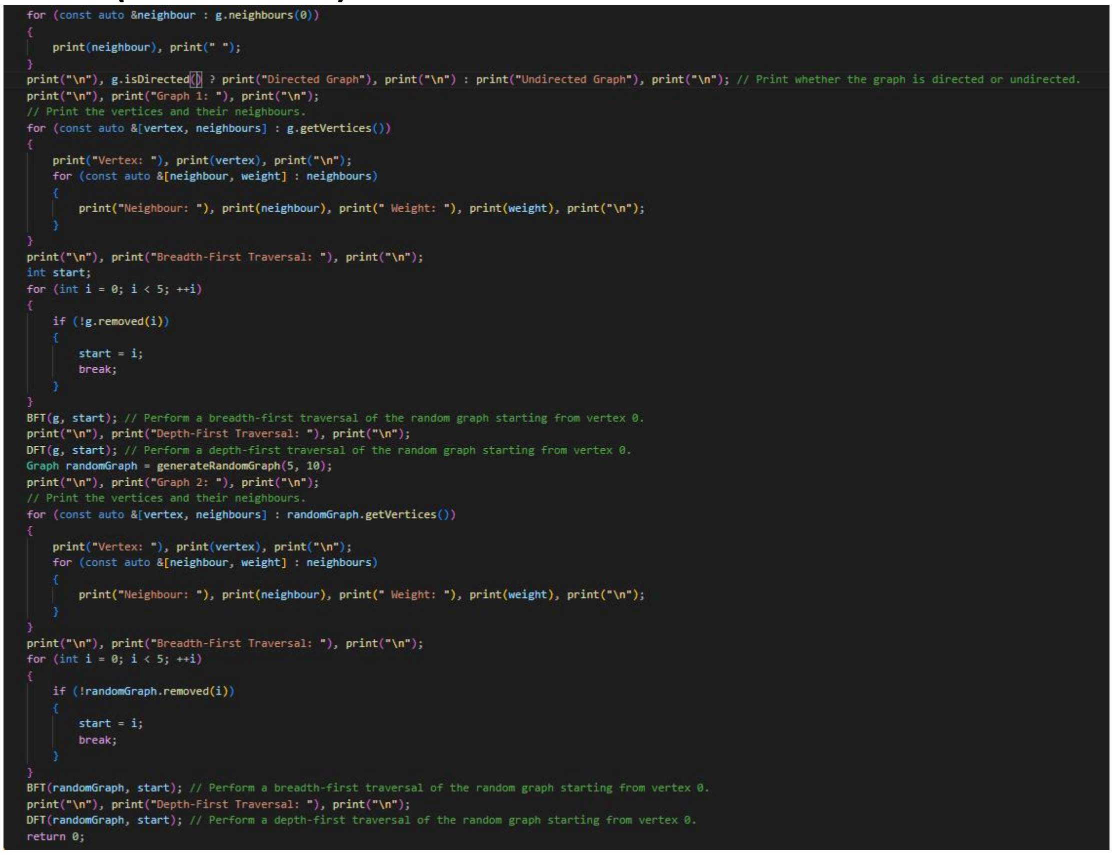
# Main.cpp

Part 1: (Please zoom)



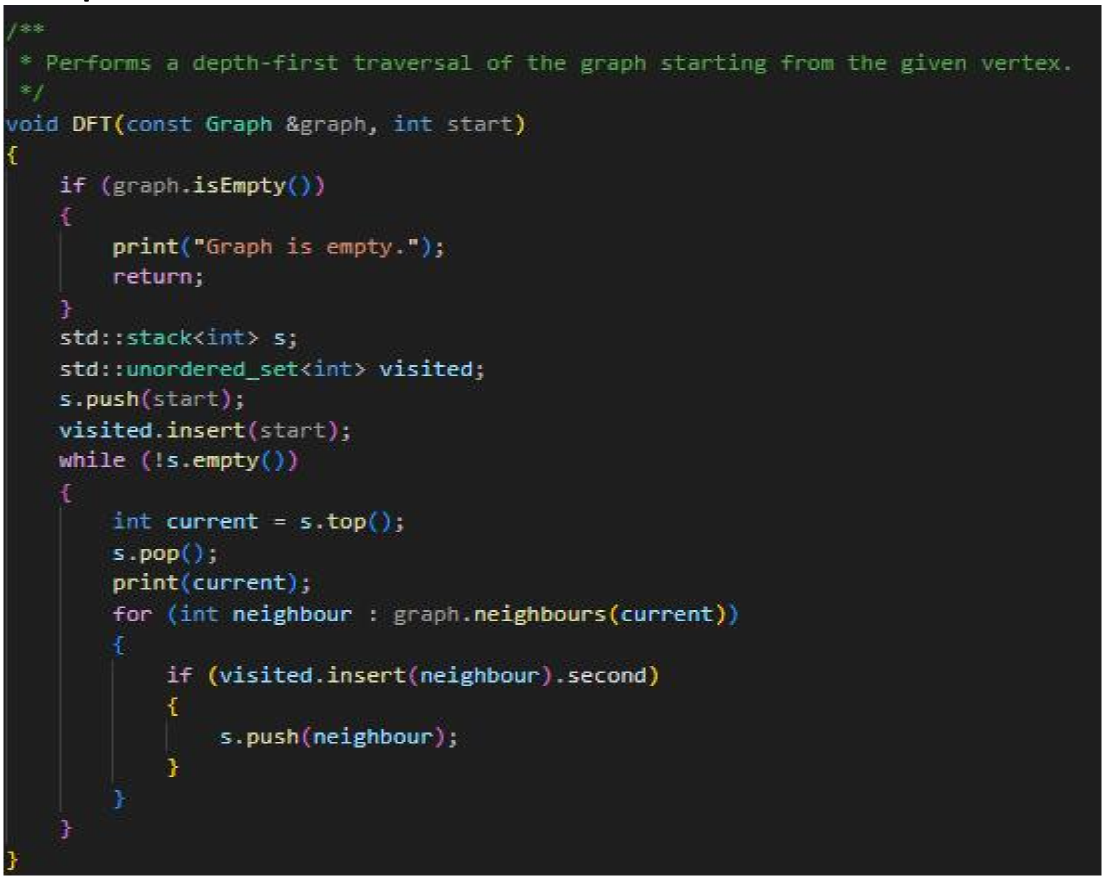
```cpp
int main()
{
    Graph g(false);
    g = generateRandomGraph(5, 10);
    print("Degree of vertex 0: "), print(g.degree(0)), print("\n"), print("Indegree of vertex 0: "), print(g.indegree(0)), print("\n"); // Print the degree of vertex 0.
    print("Outdegree of vertex 0: "), print(g.outdegree(0)), print("\n"), print("Neighbours of vertex 0: "), print("\n");            // Print the degree of vertex 0.
    for (const auto &neighbour : g.neighbours(0))
    {
        print(neighbour), print(" ");
    }
    print("\n"), print("Neighbour of vertex 0 and vertex 1: "), print(g.neighbour(0, 1)), print("\n");
    g.removeEdge(0, 1); // Remove the edge between vertex 0 and vertex 1.
    try
    {
        print("Neighbour of vertex 0 and vertex 1: "), print(g.neighbour(0, 1)), print("\n");
    }
    catch (const std::exception &e)
    {
        print(e.what()), print("\n");
    }
    g.removeVertex(0); // Remove vertex 0 from the graph.
    try
    {
        print("Degree of vertex 0: "), print(g.degree(0)), print("\n");
    }
    catch (const std::exception &e)
    {
        print(e.what()), print("\n");
    }
    try
    {
        print("Indegree of vertex 0: "), print(g.indegree(0)), print("\n");
    }
    catch (const std::exception &e)
    {
        print(e.what()), print("\n");
    }
    try
    {
        print("Outdegree of vertex 0: "), print(g.outdegree(0)), print("\n");
    }
    catch (const std::exception &e)
    {
        print(e.what()), print("\n");
    }
    for (const auto &neighbour : g.neighbours(0))
    {
        print(neighbour), print(" ");
    }
```

Part 2: (Please zoom)



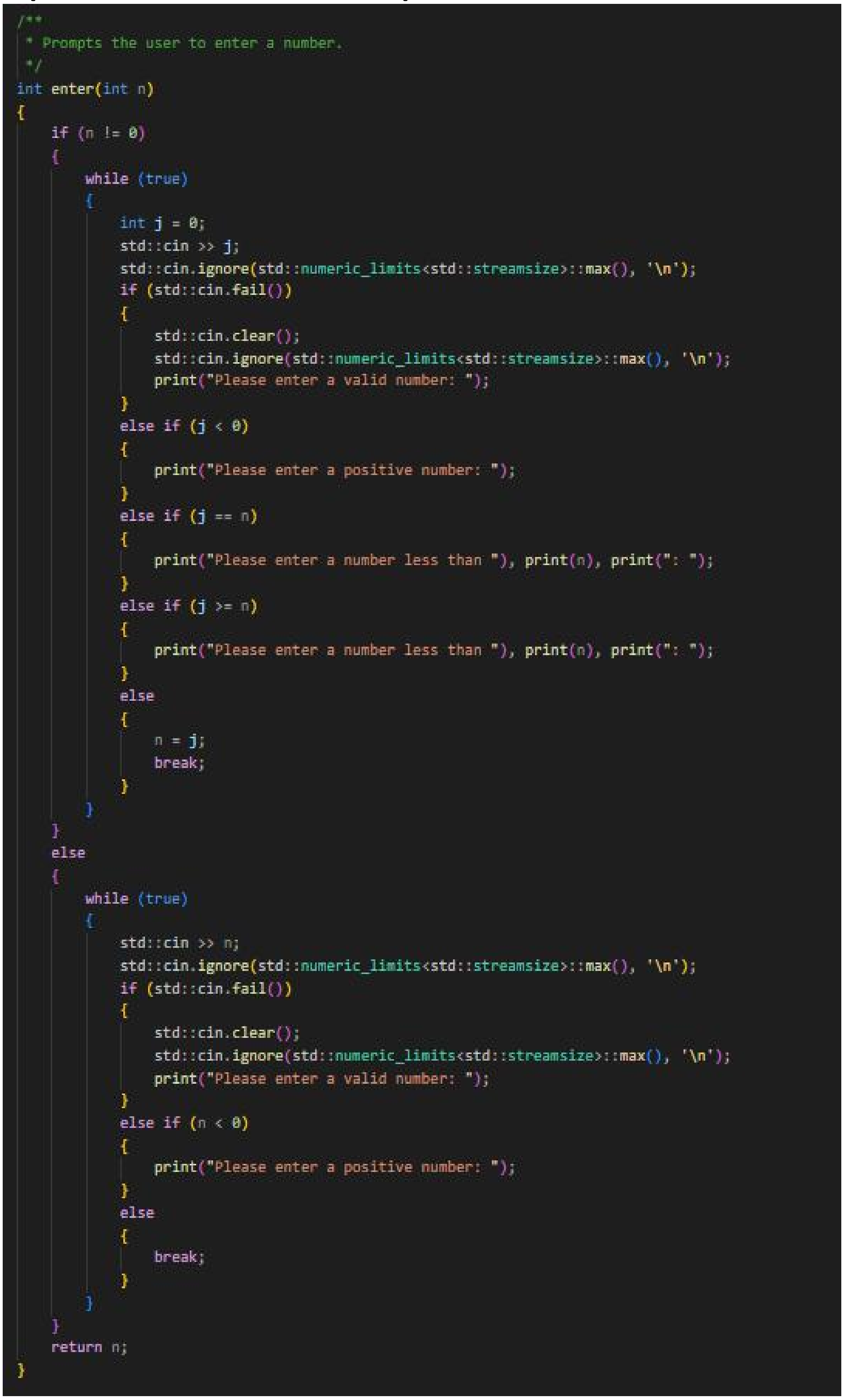
```cpp
for (const auto &neighbour : g.neighbours(0))
{
    print(neighbour), print(" ");
}
print("\n"), g.isDirected() ? print("Directed Graph"), print("\n") : print("Undirected Graph"), print("\n"); // Print whether the graph is directed or undirected.
print("\n"), print("Graph 1: "), print("\n");
// Print the vertices and their neighbours.
for (const auto &[vertex, neighbours] : g.getVertices())
{
    print("Vertex: "), print(vertex), print("\n");
    for (const auto &[neighbour, weight] : neighbours)
    {
        print("Neighbour: "), print(neighbour), print(" Weight: "), print(weight), print("\n");
    }
}
print("\n"), print("Breadth-First Traversal: "), print("\n");
int start;
for (int i = 0; i < 5; ++i)
{
    if (!g.removed(i))
    {
        start = i;
        break;
    }
}
BFT(g, start); // Perform a breadth-first traversal of the random graph starting from vertex 0.
print("\n"), print("Depth-First Traversal: "), print("\n");
DFT(g, start); // Perform a depth-first traversal of the random graph starting from vertex 0.
Graph randomGraph = generateRandomGraph(5, 10);
print("\n"), print("Graph 2: "), print("\n");
// Print the vertices and their neighbours.
for (const auto &[vertex, neighbours] : randomGraph.getVertices())
{
    print("Vertex: "), print(vertex), print("\n");
    for (const auto &[neighbour, weight] : neighbours)
    {
        print("Neighbour: "), print(neighbour), print(" Weight: "), print(weight), print("\n");
    }
}
print("\n"), print("Breadth-First Traversal: "), print("\n");
for (int i = 0; i < 5; ++i)
{
    if (!randomGraph.removed(i))
    {
        start = i;
        break;
    }
}
BFT(randomGraph, start); // Perform a breadth-first traversal of the random graph starting from vertex 0.
print("\n"), print("Depth-First Traversal: "), print("\n");
DFT(randomGraph, start); // Perform a depth-first traversal of the random graph starting from vertex 0.
return 0;
```

# Helper functions:

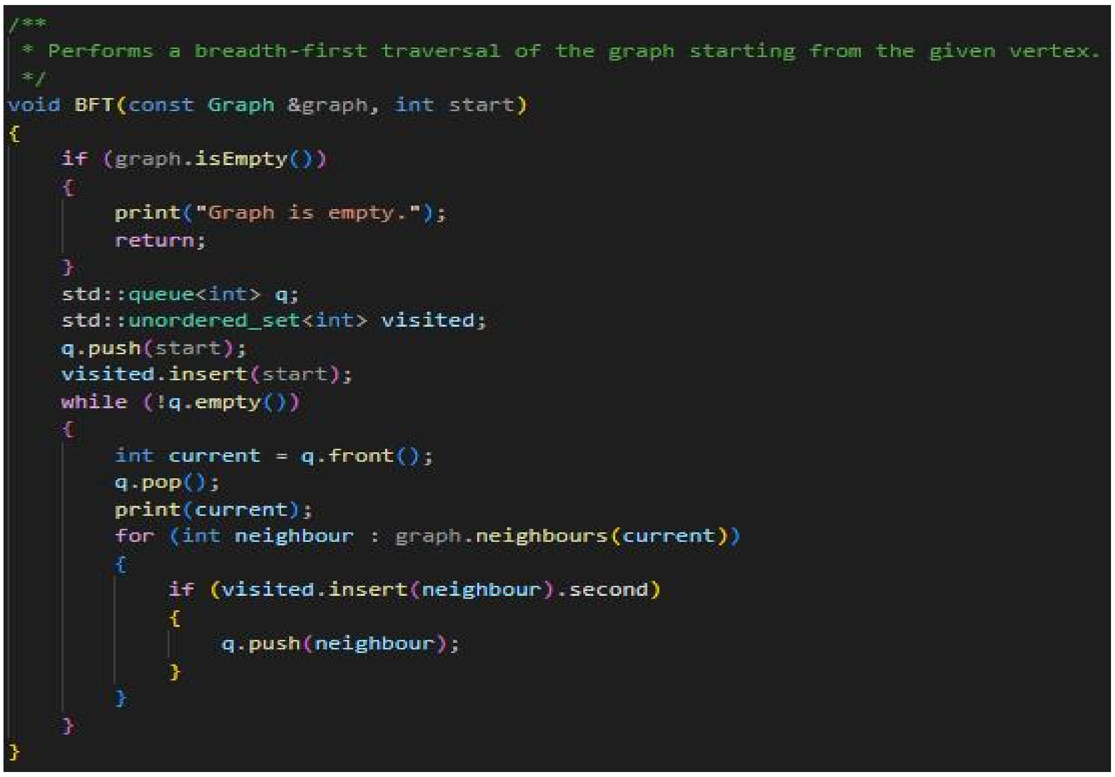


```cpp
/**
 * Performs a depth-first traversal of the graph starting from the given vertex.
 */
void DFT(const Graph &graph, int start)
{
    if (graph.isEmpty())
    {
        print("Graph is empty.");
        return;
    }
    std::stack<int> s;
    std::unordered_set<int> visited;
    s.push(start);
    visited.insert(start);
    while (!s.empty())
    {
        int current = s.top();
        s.pop();
        print(current);
        for (int neighbour : graph.neighbours(current))
        {
            if (visited.insert(neighbour).second)
            {
                s.push(neighbour);
            }
        }
    }
}
```

Input function which takes input from the user:

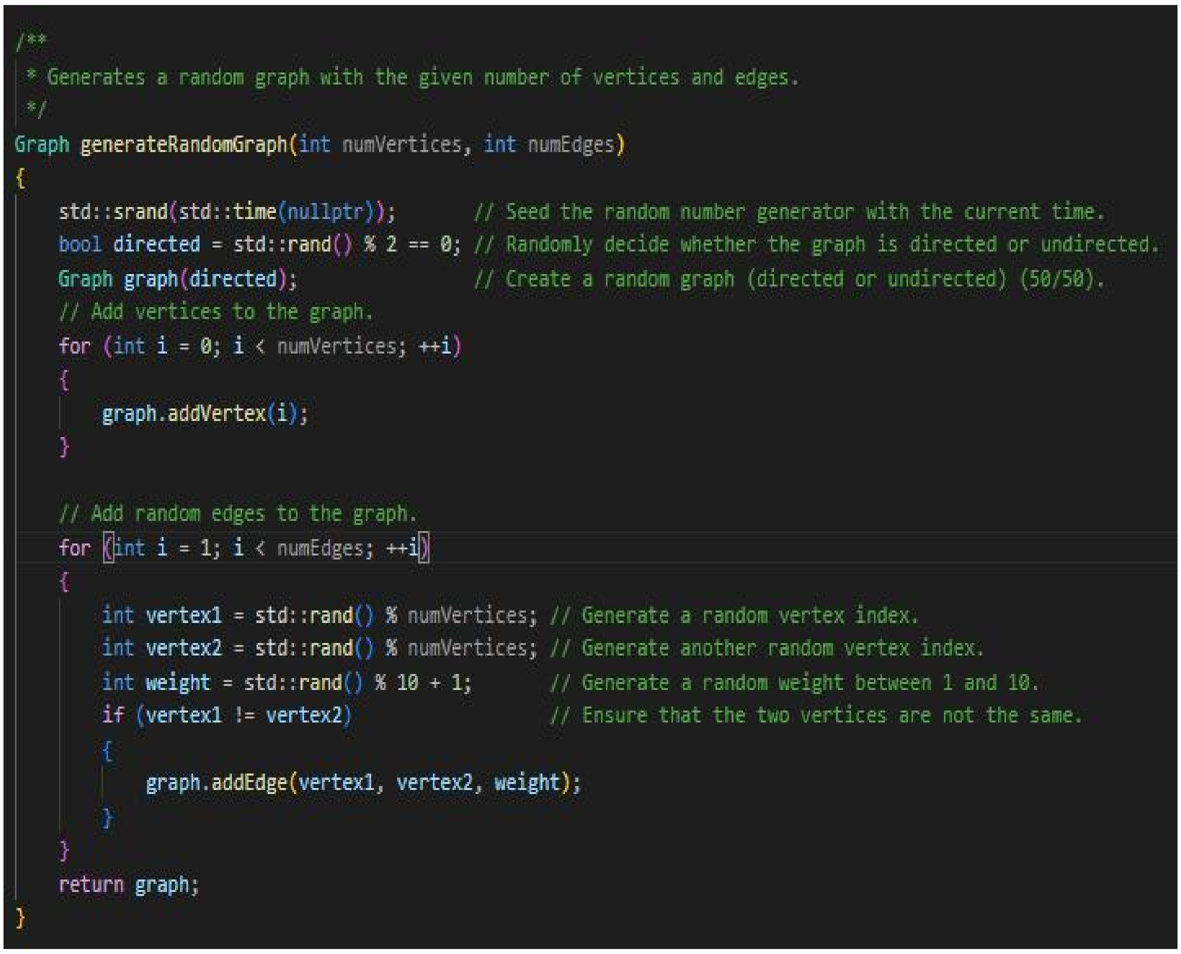


```cpp
/**
 * Prompts the user to enter a number.
 */
int enter(int n)
{
    if (n != 0)
    {
        while (true)
        {
            int j = 0;
            std::cin >> j;
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            if (std::cin.fail())
            {
                std::cin.clear();
                std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                print("Please enter a valid number: ");
            }
            else if (j < 0)
            {
                print("Please enter a positive number: ");
            }
            else if (j == n)
            {
                print("Please enter a number less than "), print(n), print(": ");
            }
            else if (j >= n)
            {
                print("Please enter a number less than "), print(n), print(": ");
            }
            else
            {
                n = j;
                break;
            }
        }
    }
    else
    {
        while (true)
        {
            std::cin >> n;
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            if (std::cin.fail())
            {
                std::cin.clear();
                std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                print("Please enter a valid number: ");
            }
            else if (n < 0)
            {
                print("Please enter a positive number: ");
            }
            else
            {
                break;
            }
        }
    }
    return n;
}
```



```cpp
/**
 * Performs a breadth-first traversal of the graph starting from the given vertex.
 */
void BFT(const Graph &graph, int start)
{
    if (graph.isEmpty())
    {
        print("Graph is empty.");
        return;
    }
    std::queue<int> q;
    std::unordered_set<int> visited;
    q.push(start);
    visited.insert(start);
    while (!q.empty())
    {
        int current = q.front();
        q.pop();
        print(current);
        for (int neighbour : graph.neighbours(current))
        {
            if (visited.insert(neighbour).second)
            {
                q.push(neighbour);
            }
        }
    }
}
```

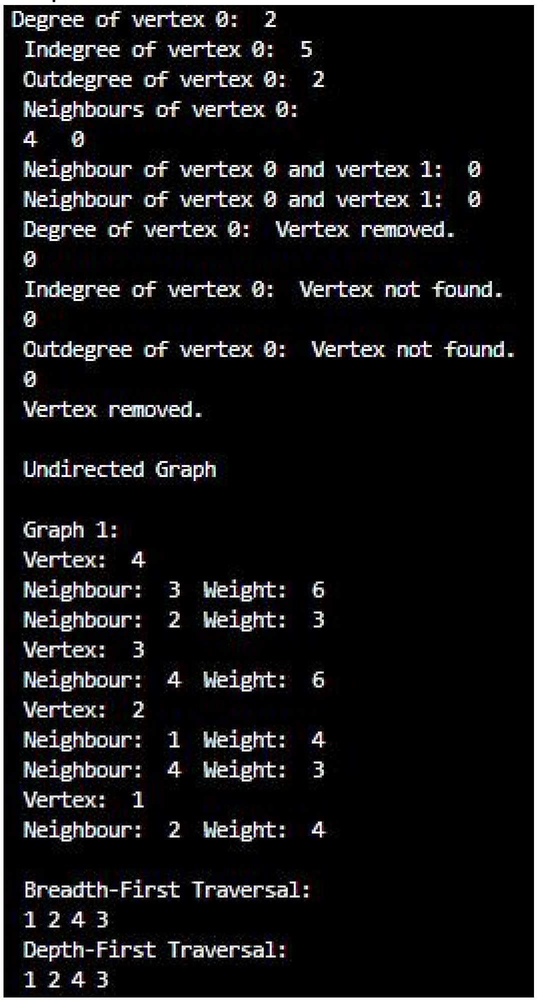
Bonus: Write a program to generate a random graph.



```cpp
/**
 * Generates a random graph with the given number of vertices and edges.
 */
Graph generateRandomGraph(int numVertices, int numEdges)
{
    std::srand(std::time(nullptr));       // Seed the random number generator with the current time.
    bool directed = std::rand() % 2 == 0; // Randomly decide whether the graph is directed or undirected.
    Graph graph(directed);                // Create a random graph (directed or undirected) (50/50).
    // Add vertices to the graph.
    for (int i = 0; i < numVertices; ++i)
    {
        graph.addVertex(i);
    }

    // Add random edges to the graph.
    for (int i = 1; i < numEdges; ++i)
    {
        int vertex1 = std::rand() % numVertices; // Generate a random vertex index.
        int vertex2 = std::rand() % numVertices; // Generate another random vertex index.
        int weight = std::rand() % 10 + 1;       // Generate a random weight between 1 and 10.
        if (vertex1 != vertex2)                  // Ensure that the two vertices are not the same.
        {
            graph.addEdge(vertex1, vertex2, weight);
        }
    }
    return graph;
}
```

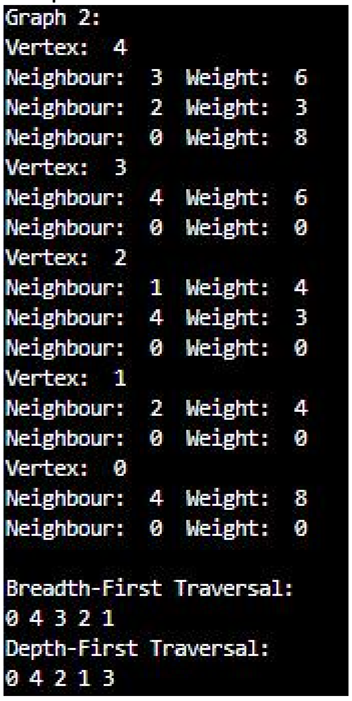
Output:

Graph 1:



```
Degree of vertex 0:  2
 Indegree of vertex 0:  5
 Outdegree of vertex 0:  2
 Neighbours of vertex 0:
 4   0
 Neighbour of vertex 0 and vertex 1:  0
 Neighbour of vertex 0 and vertex 1:  0
 Degree of vertex 0:  Vertex removed.
 0
 Indegree of vertex 0:  Vertex not found.
 0
 Outdegree of vertex 0:  Vertex not found.
 0
 Vertex removed.

 Undirected Graph

 Graph 1:
 Vertex:  4
 Neighbour:  3  Weight:  6
 Neighbour:  2  Weight:  3
 Vertex:  3
 Neighbour:  4  Weight:  6
 Vertex:  2
 Neighbour:  1  Weight:  4
 Neighbour:  4  Weight:  3
 Vertex:  1
 Neighbour:  2  Weight:  4

 Breadth-First Traversal:
 1 2 4 3
 Depth-First Traversal:
 1 2 4 3
```

Graph 2:



```
Graph 2:
Vertex:  4
Neighbour:  3  Weight:  6
Neighbour:  2  Weight:  3
Neighbour:  0  Weight:  8
Vertex:  3
Neighbour:  4  Weight:  6
Neighbour:  0  Weight:  0
Vertex:  2
Neighbour:  1  Weight:  4
Neighbour:  4  Weight:  3
Neighbour:  0  Weight:  0
Vertex:  1
Neighbour:  2  Weight:  4
Neighbour:  0  Weight:  0
Vertex:  0
Neighbour:  4  Weight:  8
Neighbour:  0  Weight:  0

Breadth-First Traversal:
0 4 3 2 1
Depth-First Traversal:
0 4 2 1 3
```

Summary:
This lab focused on implementing a Graph data structure and its associated operations. The key components included:

1. A Graph class with methods for basic graph operations.
2. Implementation of various graph functions in Graph.cpp, including:
   - Checking if the graph is empty or directed
   - Adding and removing vertices and edges
   - Counting vertices and edges
   - Calculating indegree, outdegree, and degree of vertices
   - Finding neighbors of vertices
   - Traversing algorithms

3. A Main.cpp file demonstrating the usage of the Graph class, including:
   - Creating and manipulating two different graphs
   - Performing various operations on these graphs

4. Helper functions for input handling and graph manipulation.
5. A bonus program to generate a random graph.

The lab provided practical experience in working with graph data structures, implementing common graph algorithms, and understanding the relationships between vertices and edges in both directed and undirected graphs. The output screenshots demonstrate the successful implementation and testing of the graph operations.