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

struct nodeGraph{
    int vertex;
    nodeGraph* next;
};

//declaring
bool graphCycles = false;  // Flag to determine if graphCycles exist in the graph
bool* VisitedNode;  // Array to track visited nodes
int i, j, k;
int numberOfEdges;  // Number of edges
int numberOfVertices;  // Number of vertices

class Graph{
```

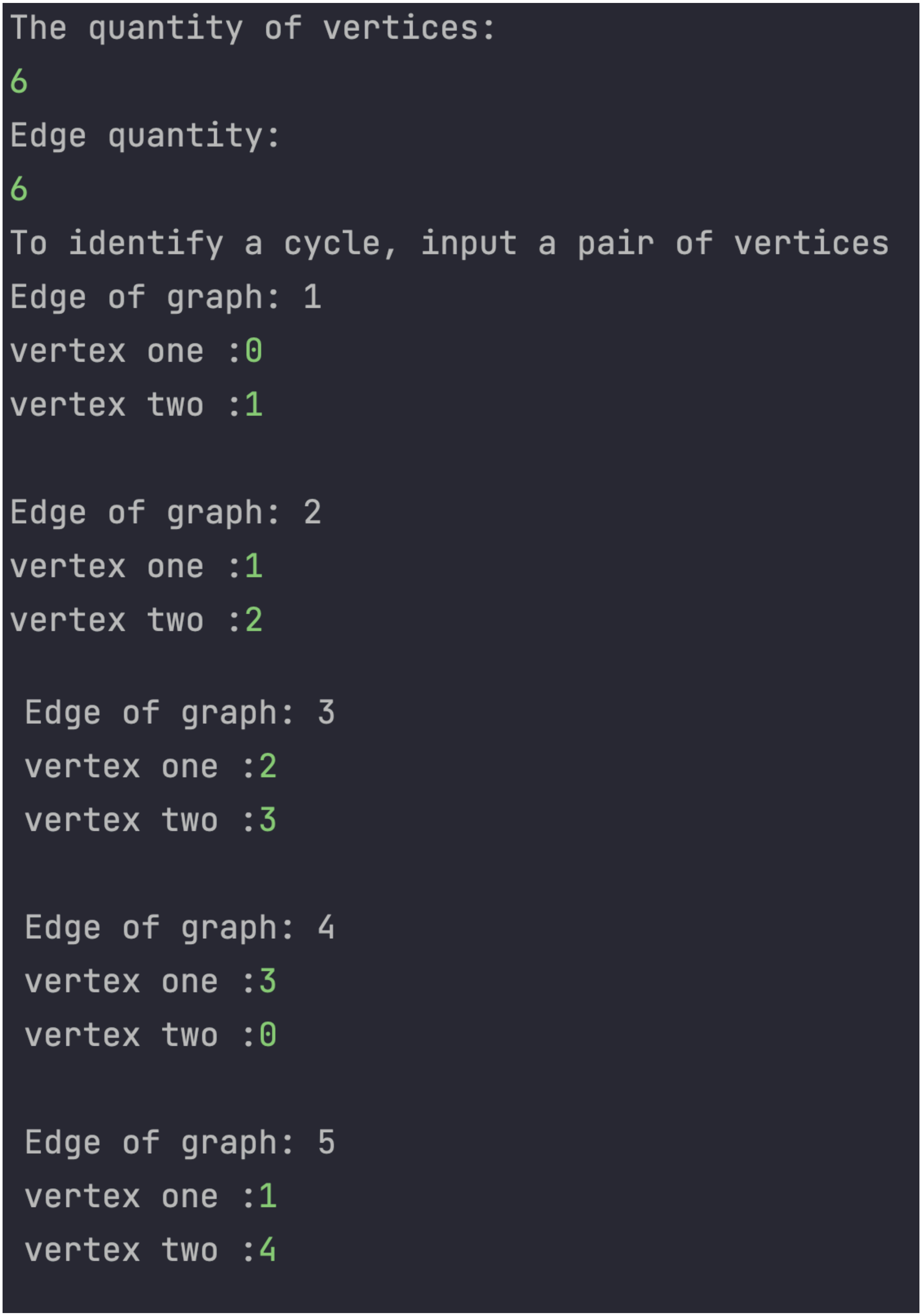
```
private:
    int vertexOne;
    int vertexTwo;

public:
    nodeGraph* headNodes;  // Array of nodes representing the graph
    Graph(int nodes)  // Constructor to initialize the graph
    {
        numberOfVertices = nodes;
        headNodes = new nodeGraph[numberOfVertices];  // Allocate memory for array of nodes
        for (i = 0; i < numberOfVertices; i++)
        {
            headNodes[i].vertex = i;  // Assign each nodeGraph a unique vertex number
            headNodes[i].next = nullptr;  // Initialize the 'next' pointer of each nodeGraph to null
        }
    }

    void create()
    {
        nodeGraph* prevNode;
        nodeGraph* newNode;
        cout << "Edge quantity: " << endl;
        cin >> numberOfEdges;
        cout << "To identify a cycle, input a pair of vertices\n";
        for (i = 1; i <= numberOfEdges; i++)
        {
            cout << "Edge of graph: " << i << "\nvertex one :";
            cin >> vertexOne;
            cout << "vertex two :";
            cin >> vertexTwo;
            cout << endl;

            // Creating edges by updating the adjacency list representation
            newNode = new nodeGraph;
            newNode->vertex = vertexTwo;

            if (headNodes[vertexOne].next == nullptr)
            {
                newNode->next = nullptr;
                headNodes[vertexOne].next = newNode;
            }
            else{
                prevNode = &headNodes[vertexOne];
                while (prevNode->next != nullptr)
                {
                    prevNode = prevNode->next;
                }
                newNode->next = nullptr;
                prevNode->next = newNode;
            }
```

```
            newNode = new nodeGraph;
            newNode->vertex = vertexOne;
```

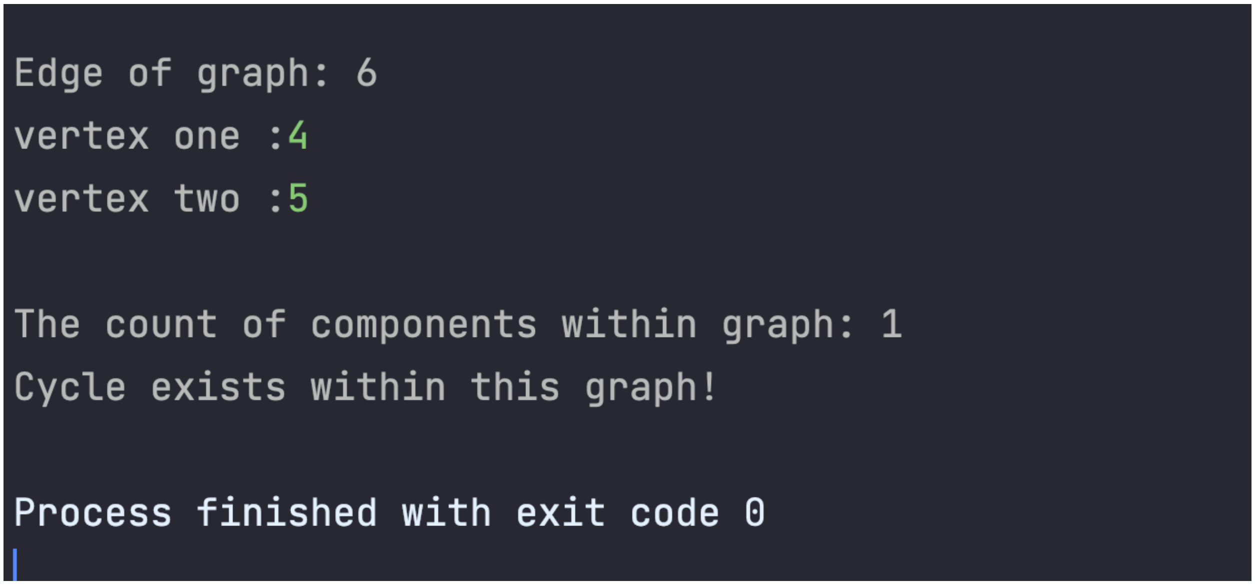
```
            if (headNodes[vertexTwo].next == nullptr)
            {
                newNode->next = nullptr;
                headNodes[vertexTwo].next = newNode;
            }
            else
            {
                prevNode = &headNodes[vertexTwo];
                while (prevNode->next != nullptr)
                {
                    prevNode = prevNode->next;
                }
                newNode->next = nullptr;
                prevNode->next = newNode;
            }
        }
    }
```

```
    void depthFirstSearch(int father, int v){
        VisitedNode[v] = true;  // Mark the current nodeGraph as visited
        nodeGraph* adjNode = headNodes[v].next;  // Get the adjacent nodes of the current nodeGraph
        while (adjNode)
        {
            if (!VisitedNode[adjNode->vertex])
            {
                depthFirstSearch(v, adjNode->vertex);  // Recursive depthFirstSearch call for unvisited nodes
            }
            else if (father != adjNode->vertex)
```

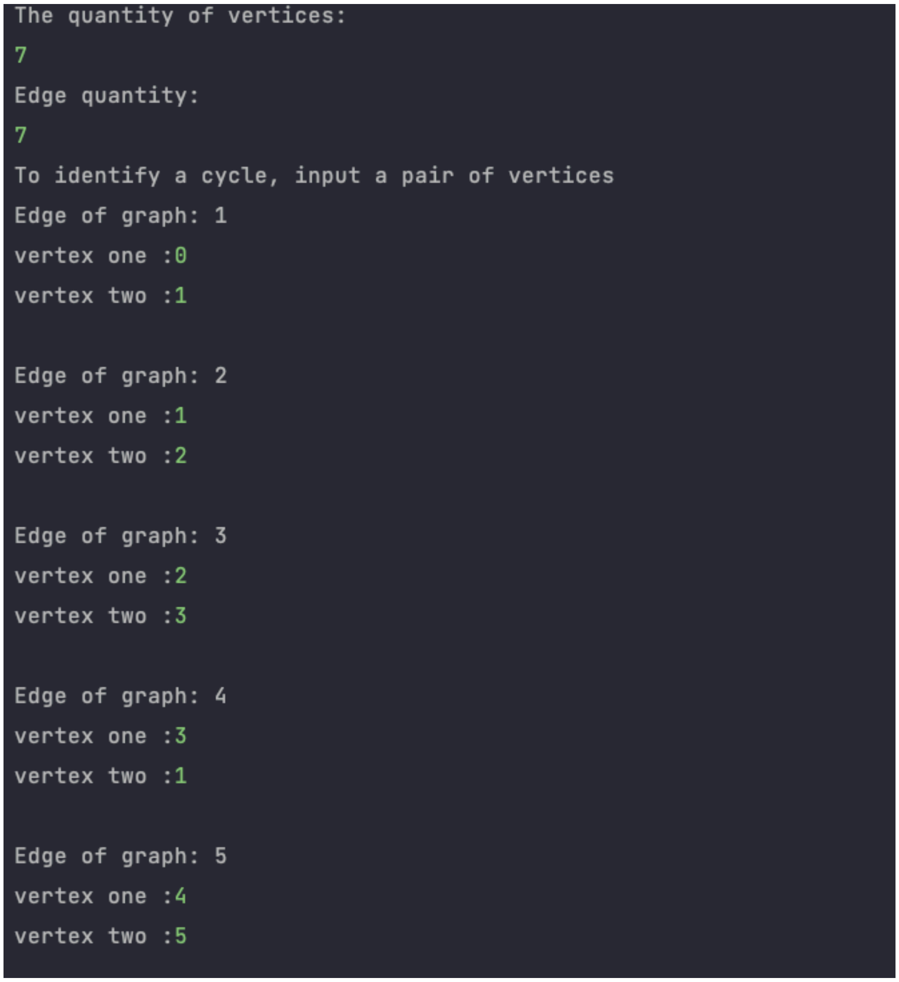
```cpp
            {
                graphCycles = true;  // Detect cycle if the adjacent nodeGraph is already visited and not the father
            }
            adjNode = adjNode->next;  // Move to the next adjacent nodeGraph
        }
    }
};

int main()
{
    cout << "The quantity of vertices: " << endl;
    cin >> numberOfVertices;
    VisitedNode = new bool[numberOfVertices];  // Allocate memory for the visited array
    int numberOfComponents = 0;  // Count of connected components

    Graph G(numberOfVertices);  // Create a graph object with 'numberOfVertices' vertices
    G.create();  // Create the graph by inputting edges
    for (i = 0; i <= numberOfVertices; i++){
        VisitedNode[i] = false; } // Initialize visited array for all nodes as false
    for (i = 0; i < numberOfVertices; i++){
        VisitedNode[i] = false;
        for (j = 0; j < numberOfVertices; j++){
            if (!VisitedNode[j]){
                G.depthFirstSearch(0, j);  // Perform depthFirstSearch on unvisited nodes
                numberOfComponents++; } // Increment the count of connected components
        }
        cout << "The count of components within graph: " << numberOfComponents << endl;
        if (graphCycles)
            cout << "Cycle exists within this graph!\n";
        else
            cout << "No cycle present in this graph\n";
        return 0;
    }
}
```



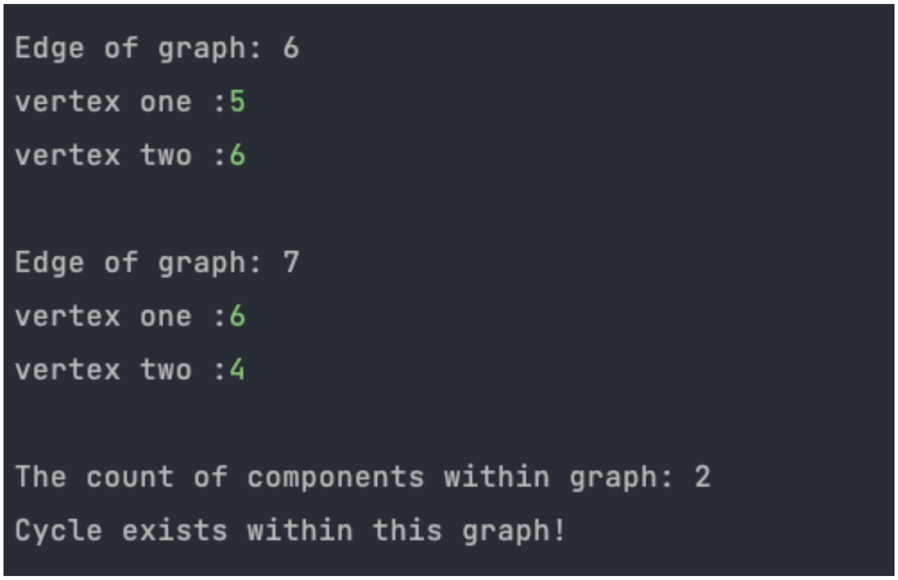
```
The quantity of vertices:
6
Edge quantity:
6
To identify a cycle, input a pair of vertices
Edge of graph: 1
vertex one :0
vertex two :1

Edge of graph: 2
vertex one :1
vertex two :2

 Edge of graph: 3
 vertex one :2
 vertex two :3

 Edge of graph: 4
 vertex one :3
 vertex two :0

 Edge of graph: 5
 vertex one :1
 vertex two :4
```



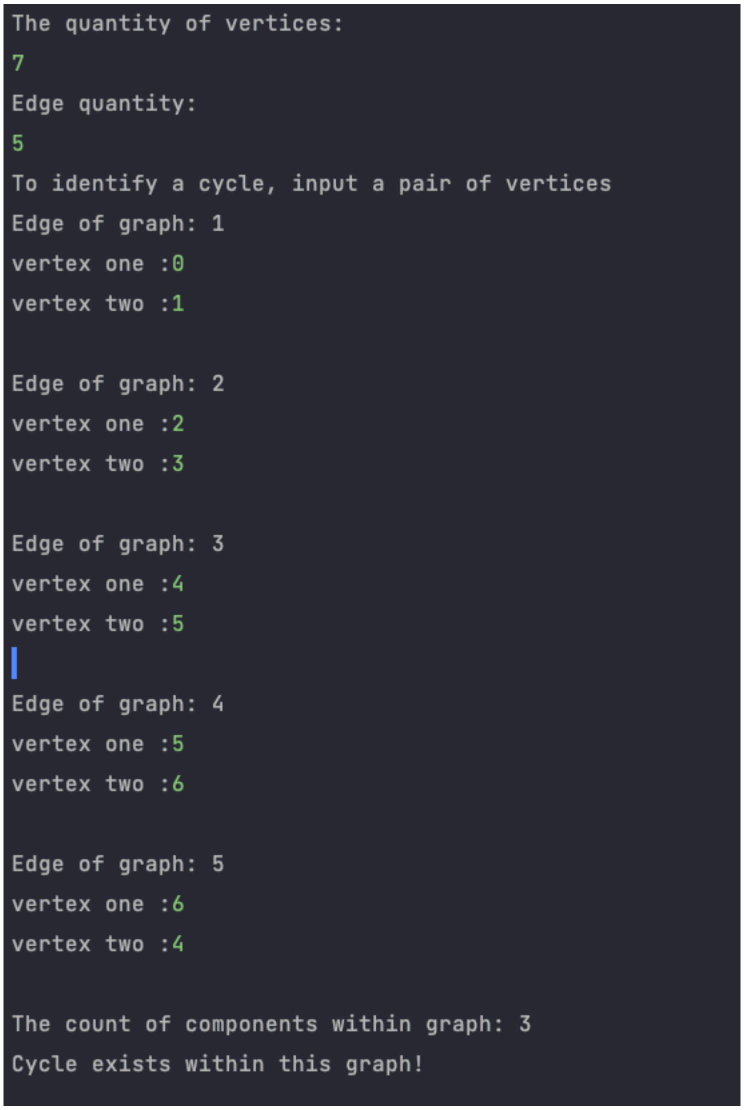
```
Edge of graph: 6
vertex one :4
vertex two :5

The count of components within graph: 1
Cycle exists within this graph!

Process finished with exit code 0
```



```
The quantity of vertices:
7
Edge quantity:
7
To identify a cycle, input a pair of vertices
Edge of graph: 1
vertex one :0
vertex two :1

Edge of graph: 2
vertex one :1
vertex two :2

Edge of graph: 3
vertex one :2
vertex two :3

Edge of graph: 4
vertex one :3
vertex two :1

Edge of graph: 5
vertex one :4
vertex two :5
```



```
Edge of graph: 6
vertex one :5
vertex two :6

Edge of graph: 7
vertex one :6
vertex two :4

The count of components within graph: 2
Cycle exists within this graph!
```



```
The quantity of vertices:
7
Edge quantity:
5
To identify a cycle, input a pair of vertices
Edge of graph: 1
vertex one :0
vertex two :1

Edge of graph: 2
vertex one :2
vertex two :3

Edge of graph: 3
vertex one :4
vertex two :5

Edge of graph: 4
vertex one :5
vertex two :6

Edge of graph: 5
vertex one :6
vertex two :4

The count of components within graph: 3
Cycle exists within this graph!
```