Code Structure

Includes and Namespaces:

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

1. This section includes necessary libraries and uses the standard namespace.

Type Definitions:

```
 typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
```

2. It defines shorthand for commonly used types like pairs of integers and vectors.

Global Variables:

```
 vector<vii> AdjList;
priority_queue<ii, vector<ii>, greater<ii>> pq;
vi dist;
#define INF 1000000000
```

3.
    - `AdjList` stores the adjacency list where each node points to a list of pairs representing connected nodes and their weights.
    - `pq` is a priority queue that uses a min-heap to store vertices according to their shortest known distance.
    - `dist` holds the shortest distance from the source to each vertex.
    - `INF` is a large number representing infinity.

Dijkstra's Algorithm Function:

```
 void dijkstra(int s){
    dist[s] = 0;
    pq.push(ii(0, s));
    while (!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue;
        for (int j = 0; j < AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
}
```

4. This function updates the shortest paths from a source node s to all other nodes in the graph using a greedy method. Nodes are processed according to their distance from the source, starting with the closest.

Main Function:

```
 int main() {
    int V, E, s, a, b, w;
    cin >> V >> E >> s;
    AdjList.assign(V, vii());
    dist.assign(V, INF);
    for (int i = 0; i < E; i++) {
        cin >> a >> b >> w;
        AdjList[a].push_back(ii(b, w));
    }
    dijkstra(s);
    for (int i = 0; i < V; i++)
        cout << "SP(" << s << "," << i << ")=" << dist[i] << endl;
    return 0;
}
```

5.
    - Inputs the number of vertices `V`, edges `E`, and the source vertex `s`.
    - Constructs the graph and initializes distances.
    - Executes Dijkstra's algorithm.
    - Outputs the shortest path to each vertex from the source.

Dijkstra's Algorithm Explained

Dijkstra's algorithm works by maintaining a priority queue (min-heap) that prioritizes the vertex with the shortest tentative distance:

1. **Initialization:**
    - Set the distance of the source vertex to 0 and all others to infinity.
    - Push the source vertex into the queue.
2. **Loop Until Queue is Empty:**
    - Extract the vertex `u` with the smallest distance from the priority queue.
    - For each adjacent vertex `v` of `u`, if the path through `u` offers a shorter distance than the current known distance to `v`, update `v`'s distance and push it into the queue.
3. **Relaxation:**
    - The process of updating the distance of adjacent vertices is known as relaxation. If `dist[u] + weight(u, v) < dist[v]`, then update `dist[v]` and push `v` into the queue.

Dijkstra's algorithm efficiently finds the shortest path in graphs with non-negative edge weights and has a time complexity ranging from O((V+E)logV)O((V + E) \log V) using a binary heap.

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

vector<vii> AdjList;
vi dist;
#define INF 1000000000

void dijkstra(int s) {
    int V = AdjList.size();
    dist.assign(V, INF);
    set<ii> pq;  // Set to maintain the order of distances
    dist[s] = 0;
    pq.insert({0, s});

    while (!pq.empty()) {
        auto [d, u] = *pq.begin();
        pq.erase(pq.begin());  // Remove the top element

        if (d > dist[u]) continue;

        for (auto [v, w] : AdjList[u]) {
            if (dist[u] + w < dist[v]) {
                pq.erase({dist[v], v});  // Remove the old distance
                dist[v] = dist[u] + w;
                pq.insert({dist[v], v});  // Insert the new distance
            }
        }
    }
}

int main() {
    int V, E, s, a, b, w;
    cin >> V >> E >> s;
    AdjList.assign(V, vii());

    for (int i = 0; i < E; i++) {
        cin >> a >> b >> w;
        AdjList[a].push_back({b, w});
        AdjList[b].push_back({a, w}); // If the graph is undirected
    }

    dijkstra(s);

    for (int i = 0; i < V; i++)
        cout << "SP(" << s << "," << i << ")=" << dist[i] << endl;

    return 0;
}
```