创建带有权重的无向图：

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

using namespace std;

// 边的结构体
struct Edge {
    int to;
    int weight;

    Edge(int t, int w) : to(t), weight(w) {}
};

// 图的结构体
struct Graph {
    vector<vector<Edge>> adjList;

    Graph(int n) : adjList(n) {}

    // 添加无向边
    void addEdge(int from, int to, int weight) {
        adjList[from].emplace_back(to, weight);
        adjList[to].emplace_back(from, weight);
    }
};

int main() {
    // 创建图并添加带有权重的边
    Graph weightedGraph(5);
    weightedGraph.addEdge(0, 1, 2);
    weightedGraph.addEdge(0, 3, 3);
    weightedGraph.addEdge(1, 2, 4);
    weightedGraph.addEdge(1, 3, 4);
    weightedGraph.addEdge(1, 4, 3);
    weightedGraph.addEdge(2, 4, 5);
    weightedGraph.addEdge(3, 4, 1);

    // 输出图的边及其权重
    cout << "图的边及权重:" << endl;
    for (int i = 0; i < weightedGraph.adjList.size(); ++i) {
        for (const Edge& edge : weightedGraph.adjList[i]) {
            cout << "Edge: " << i << " -- " << edge.to << " Weight: " <<
edge.weight << endl;
        }
    }

    return 0;
}
```

`Graph` 结构体表示图，而 `Edge` 结构体表示图中的边，包括边的目标顶点和权重。在 `main` 函数中，创建了一个带有权重的无向图。

以下是使用Prim算法求解最小生成树的C++代码，带有权重的无向图：

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

using namespace std;

// 边的结构体
struct Edge {
    int to;
    int weight;

    Edge(int t, int w) : to(t), weight(w) {}
};

// 图的结构体
struct Graph {
    vector<vector<Edge>> adjList;

    Graph(int n) : adjList(n) {}

    // 添加无向边
    void addEdge(int from, int to, int weight) {
        adjList[from].emplace_back(to, weight);
        adjList[to].emplace_back(from, weight);
    }
};

// Prim算法
vector<Edge> prim(const Graph& graph) {
    int n = graph.adjList.size();
    int startNode = 0;

    // 存储已经访问的节点
    set<int> visited;

    // 优先队列，用于存储边的权重和连接的节点
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

    // 最小生成树的边
    vector<Edge> mstEdges;

    // 将起始节点加入访问集合
    visited.insert(startNode);

    // 将起始节点的所有边加入优先队列
    for (const Edge& edge : graph.adjList[startNode]) {
        pq.push({edge.weight, edge.to});
    }

    while (!pq.empty()) {
        int weight = pq.top().first;
```

```
        int current = pq.top().second;
        pq.pop();

        if (visited.find(current) == visited.end()) {
            visited.insert(current);

            for (const Edge& edge : graph.adjList[current]) {
                if (visited.find(edge.to) == visited.end()) {
                    pq.push({edge.weight, edge.to});
                }
            }

            mstEdges.emplace_back(current, weight);
        }
    }

    return mstEdges;
}

int main() {
    // 创建带有权重的无向图
    Graph weightedGraph(5);
    weightedGraph.addEdge(0, 1, 2);
    weightedGraph.addEdge(0, 3, 3);
    weightedGraph.addEdge(1, 2, 4);
    weightedGraph.addEdge(1, 3, 4);
    weightedGraph.addEdge(1, 4, 3);
    weightedGraph.addEdge(2, 4, 5);
    weightedGraph.addEdge(3, 4, 1);

    // 使用Prim算法求解最小生成树
    vector<Edge> result = prim(weightedGraph);

    // 输出最小生成树的边及其权重
    cout << "最小生成树的边及权重:" << endl;
    for (const Edge& edge : result) {
        cout << "Edge: " << edge.to << " Weight: " << edge.weight << endl;
    }

    return 0;
}
```

这个代码在 `main` 函数中创建了一个带有权重的无向图，并使用Prim算法找到了最小生成树的边及其权重。你可以根据需要进行修改和扩展。

最小生成树（Minimum Spanning Tree，MST）是一个连通图的生成树（即树形的子图），它包含了图中的所有顶点，并且边的权重之和最小。**生成树是无环的**，并且具有最小的总权重。在图论中，最小生成树是解决网络设计、通信网络布线、电缆布线等问题的重要工具。

有两个主要的算法用于找到图的最小生成树：Prim算法和Kruskal算法。

Prim算法：

1. **初始化：** 选择图中的一个起始节点，将其加入最小生成树中。将与这个节点相邻的边的权重和目标节点加入优先队列。
2. **迭代：** 从优先队列中选择具有最小权重的边，将其目标节点加入最小生成树。将与新节点相邻的边的权重和目标节点加入优先队列。
3. **重复：** 重复上述步骤，直到最小生成树包含了图中的所有顶点。

Kruskal算法：

1. **初始化：** 将图中的所有边按照权重从小到大排序。
2. **迭代：** 依次考虑排好序的边，如果一条边的两个端点不在同一连通分量中，就将这条边加入最小生成树，并合并这两个连通分量。
3. **重复：** 重复上述步骤，直到最小生成树包含了图中的所有顶点。

这两种算法都保证生成的树是无环的、连通的，并且具有最小的总权重。选择使用哪种算法通常取决于具体的应用和图的特性。在实际应用中，算法的选择可能基于图的稠密程度、边的权重分布以及可用的数据结构。

Prim算法是一种用于求解图的最小生成树的贪心算法。它以一个初始节点开始，逐步向最小生成树中添加边，每次选择具有最小权重的边，并确保所选边的两个端点中至少有一个是已经加入最小生成树的节点。Prim算法的基本思想是通过不断扩展当前的最小生成树，最终得到包含所有顶点的最小生成树。

以下是Prim算法的基本步骤：

1. **选择初始节点：** 从图中选择一个起始节点作为最小生成树的起点。
2. **初始化：** 将起始节点标记为已访问，并将与这个节点相邻的边的权重和目标节点加入优先队列。
3. **迭代：** 从优先队列中选择具有最小权重的边，将其目标节点标记为已访问，并将与新节点相邻的边的权重和目标节点加入优先队列。
4. **重复：** 重复上述步骤，直到最小生成树包含了图中的所有顶点。

Prim算法的关键在于如何选择具有最小权重的边。通常使用最小堆（优先队列）来维护候选边的集合，以确保每次选择的边都是当前权重最小的。通过这种方式，Prim算法保证了每一步都选择了连接当前最小生成树和未加入最小生成树的节点的最短边。

以下是Prim算法的伪代码：

```
Prim(G):
    1. 初始化一个空的最小生成树 MST
    2. 选择任意起始节点 start
    3. 将 start 标记为已访问
    4. 将与 start 相邻的边的权重和目标节点加入最小堆 priority_queue
    5. while priority_queue 不为空:
         6. 从 priority_queue 中选择具有最小权重的边 (u, v, w)
         7. if v 没有被访问:
                8. 将 v 标记为已访问
                9. 将边 (u, v, w) 加入 MST
                10. 将与节点 v 相邻的未访问节点的边加入 priority_queue
    11. 返回 MST
```

Prim算法的时间复杂度通常为O(E log V)，其中 E 是边的数量，V 是顶点的数量。这是因为在算法执行过程中，每条边都可能会被加入和弹出最小堆一次。