Listing 1: Graph.cpp Klasse

```
// graph.cpp (Implementation of Class Graph)

#include <fstream>
#include <sstream>
#include <stdexcept>
#include <limits>
#include "graph.h"

const Graph::NodeId Graph::invalid_node = -1;
const double Graph::infinite_weight = std::numeric_limits<double>::max();


void Graph::add_nodes(NodeId num_new_nodes)
{
    _nodes.resize(num_nodes() + num_new_nodes);
}

Graph::Neighbor::Neighbor(Graph::NodeId n, double w): _id(n), _edge_weight(w) {}

Graph::Graph(NodeId num, DirType dtype): dirtype(dtype), _nodes(num) {}

void Graph::remove_edge(NodeId tail, NodeId head)
{
    if (tail >= num_nodes() or tail < 0 or head >= num_nodes() or head < 0) {
        throw std::runtime_error("Edge␣cannot␣be␣removed␣due␣to␣undefined␣
            endpoint.");
    }
    _nodes[tail].remove_neighbor(head);
    if (dirtype == Graph::undirected) {
        _nodes[head].remove_neighbor(tail);
    }
}

void Graph::add_edge(NodeId tail, NodeId head, double weight)
{
    if (tail >= num_nodes() or tail < 0 or head >= num_nodes() or head < 0) {
        throw std::runtime_error("Edge␣cannot␣be␣added␣due␣to␣undefined␣endpoint
            .");
    }
    _nodes[tail].add_neighbor(head, weight);
    if (dirtype == Graph::undirected) {
        _nodes[head].add_neighbor(tail, weight);
    }
}

void Graph::add_edge1(NodeId tail, NodeId head, double weight)
{
    if (tail >= num_nodes() or tail < 0 or head >= num_nodes() or head < 0) {
        throw std::runtime_error("Edge␣cannot␣be␣added␣due␣to␣undefined␣endpoint
            .");
    }
    _nodes[tail].add_neighbor(head, weight);
}

void Graph::Node::add_neighbor(Graph::NodeId nodeid, double weight)
{
    _neighbors.push_back(Graph::Neighbor(nodeid, weight));
}
```

```
void Graph::Node::remove_neighbor(Graph::NodeId nodeid)
{
    for (int i = 0; i < _neighbors.size(); i++) {
        if (_neighbors[i].id() == nodeid) {
            _neighbors.erase(_neighbors.begin()+i);
        }
    }
}

const std::vector<Graph::Neighbor> & Graph::Node::adjacent_nodes() const
{
    return _neighbors;
}

Graph::NodeId Graph::num_nodes() const
{
    return _nodes.size();
}

const Graph::Node & Graph::get_node(NodeId node) const
{
    if (node < 0 or node >= static_cast<int>(_nodes.size())) {
        throw std::runtime_error("Invalid nodeid in Graph::get_node.");
    }
    return _nodes[node];
}

Graph::NodeId Graph::Neighbor::id() const
{
    return _id;
}

double Graph::Neighbor::edge_weight() const
{
    return _edge_weight;
}

void Graph::print() const
{
    if (dirtype == Graph::directed) {
        std::cout << "Digraph ";
    } else {
        std::cout << "Undirected graph ";
    }
    std::cout << "with " << num_nodes() << " vertices, numbered 0,...,"
              << num_nodes() - 1 << ".\n";

    for (auto nodeid = 0; nodeid < num_nodes(); ++nodeid) {
        std::cout << "The following edges are ";
        if (dirtype == Graph::directed) {
            std::cout << "leaving";
        } else {
            std::cout << "incident to";
        }
        std::cout << " vertex " << nodeid << ":\n";
        for (auto neighbor: _nodes[nodeid].adjacent_nodes()) {
            std::cout << nodeid << " - " << neighbor.id()
                      << " weight = " << neighbor.edge_weight() << "\n";
        }
    }
}
```

```
Graph::Graph(char const * filename, DirType dtype): dirtype(dtype)
{
    std::ifstream file(filename);                              // open file
    if (not file) {
        throw std::runtime_error("Cannot open file.");
    }

    Graph::NodeId num = 0;
    std::string line;
    std::getline(file, line);                   // get first line of file
    std::stringstream ss(line);                 // convert line to a stringstream
    ss >> num;                                  // for which we can use >>
    if (not ss) {
        throw std::runtime_error("Invalid file format.");
    }
    add_nodes(num);

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        Graph::NodeId head, tail;
        ss >> tail >> head;
        if (not ss) {
            throw std::runtime_error("Invalid file format.");
        }
        double weight = 1.0;
        ss >> weight;
        if (tail != head) {
            add_edge(tail, head, weight);
        }
        else {
            throw std::runtime_error("Invalid file format: loops not allowed.");
        }
    }
}
```

Listing 2: Graph.h Header

```
// graph.h (Declaration of Class Graph)
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <vector>


class Graph {
public:
    using NodeId = int;  // vertices are numbered 0,...,num_nodes()-1

    class Neighbor {
    public:
        Neighbor(Graph::NodeId n, double w);
        double edge_weight() const;
        Graph::NodeId id() const;
    private:
        Graph::NodeId _id;
        double _edge_weight;
    };

    class Node {
    public:
        void add_neighbor(Graph::NodeId nodeid, double weight);
        void remove_neighbor(Graph::NodeId id);
        const std::vector<Neighbor> & adjacent_nodes() const;
    private:
        std::vector<Neighbor> _neighbors;
    };

    enum DirType {directed, undirected};  // enum defines a type with possible
        values
    Graph(NodeId num_nodes, DirType dirtype);
    Graph(char const* filename, DirType dirtype);

    void add_nodes(NodeId num_new_nodes);
    void add_edge(NodeId tail, NodeId head, double weight = 1.0);
    void add_edge1(NodeId tail, NodeId head, double weight = 1.0);
    void remove_edge(NodeId start,NodeId end);



    NodeId num_nodes() const;
    const Node & get_node(NodeId) const;
    void print() const;

    const DirType dirtype;
    static const NodeId invalid_node;
    static const double infinite_weight;

private:
    std::vector<Node> _nodes;
};

#endif // GRAPH_H
```