```
    1: #include "graph.h"
    2: #include <algorithm>
    3: #include <cassert>
    4: #include <fstream>
    5: using namespace std;
    6:
    7: [[maybe_unused]] graph::graph(const string &file_name)
    8:     : _edges(0), _vertices(), _maxvert(-1)             // graph_2
    9: {
   10:     ifstream fin(file_name);           // Oeffne das File im ASCII-Mo
dus
   11:     if ( fin.is_open() ) {             // File gefunden:
   12: //        _edges.clear();               // Vektor leeren
   13:         unsigned int k, l;
   14:         while ( fin >> k >> l) {_edges.push_back({k, l});} // Einlese
n
   15:         if (!fin.eof()) {
   16:             // Fehlerbehandlung
   17:             cout << " Error handling \n";
   18:             if ( fin.bad() )  {throw runtime_error("Schwerer Fehler i
n istr");}
   19:             if ( fin.fail() ) { // Versuch des Aufraeumens
   20:                 cout << " Failed in reading all data.\n";
   21:                 fin.clear();
   22:             }
   23:         }
   24:         _edges.shrink_to_fit();
   25:     }
   26:     else {                             // File nicht gefunden:
   27:         cout << "\nFile " << file_name << " has not been found.\n\n"
;
   28:         assert( fin.is_open() && "File not found." );       // exepti
on handling for the poor programmer
   29:     }
   30:
   31:     DetermineNumberVertices();
   32: }
   33:
   34:
   35: vector<vector<unsigned int>> graph::get_node2nodes() const
   36: {
   37: //    size_t nnode=Nvertices();
   38:     size_t nnode = Max_vertex() + 1;      // graph_2
   39:
   40:     // Determine the neighborhood for each vertex
   41:     vector<vector<unsigned int>> n2n(nnode);
   42:     for (auto _edge : _edges) {
   43:         auto const v0 = _edge[0];
   44:         auto const v1 = _edge[1];
   45:         n2n.at(v0).push_back(v1);         // add v1 to neighborhood o
f v0
   46:         n2n.at(v1).push_back(v0);         //     and vice versa
   47:     }
   48:     // ascending sort of entries per node
   49:     for (auto & k : n2n) {
   50:         sort(k.begin(), k.end());
   51:     }
```

```
    52:
    53:
    54:     return n2n;
    55: }
    56: // graph_2
    57: void graph::DetermineNumberVertices()
    58: {
    59:     // we assume that the nodes are numbered consecutively from 0 to
n-1
    60:     // determine number of nodes
    61:     _vertices.clear();
    62:     unsigned int nnode = 0;
    63:     for (auto & _edge : _edges) {
    64:         for (unsigned int & j : _edge) {
    65:             nnode = max(nnode, j);
    66:             _vertices.insert(j);       // graph_2
    67:         }
    68:     }
    69:     if ( !_edges.empty() )    // at least one edge in graph?
    70:        {_maxvert = nnode;}
    71:     else
    72:        {_maxvert=-1;}
    73: }
    74:
    75: ostream &operator<<(ostream &s, graph const &rhs)
    76: {
    77:     s << "Graph with  " << rhs.Nedges() << " edges and  " << rhs.Nver
tices() << " vertices" << endl;
    78:
    79:     const auto &edges = rhs._edges;
    80:     s << "\n --  Edges  --\n";
    81:     for (size_t k = 0; k < edges.size(); ++k) {
    82:         s << k << " : ";
    83:         for (unsigned int j : edges[k]) {
    84:             s << j << " ";
    85:         }
    86:         s << endl;
    87:     }
    88:
    89:     s << "\n --  Vertices  --\n";              // graph_2
    90:     for (auto v : rhs._vertices) {             // graph_2
    91:         s << v << "  ";
    92:     }
    93:     s << endl;
    94:
    95:     return s;
    96: }
    97:
    98:
    99: [[maybe_unused]] bool graph::Append(unsigned int v1, unsigned int v2)
   // graph_3
   100: {
   101:     const auto ip = find(_edges.cbegin(), _edges.cend(), Edge{v1, v2}
);
   102:     bool edgeFound(ip == _edges.cend());     // really a new edge
   103:     if (edgeFound) {
   104:         _edges.push_back(Edge{v1, v2});
```

```
  105:         _vertices.insert(v1);
  106:         _maxvert = max(_maxvert, v1);
  107:         _vertices.insert(v2);
  108:         _maxvert = max(_maxvert, v2);
  109:     }
  110:     return edgeFound;
  111: }
  112:
  113: bool graph::Delete(Edge const& e)                          // graph_3
  114: {
  115:     const auto ip = find(_edges.cbegin(), _edges.cend(), e);
  116:     bool edgeFound(ip != _edges.cend());     //  edge found
  117:     if (edgeFound) {
  118:         _edges.erase(ip);
  119:         DetermineNumberVertices(); // updates _vertices, _maxvert
  120:     }
  121:     return edgeFound;
  122: }
  123:
  124: [[maybe_unused]] bool graph::Delete(unsigned int v1, unsigned int v2)
   // graph_3
  125: {
  126:     return Delete(Edge{v1, v2});
  127: }
  128:
  129: void graph::Delete(vector<Edge> const &v)                  // graph_3
  130: {
  131:     for (const auto &e : v) {
  132:         const auto ip = find(_edges.cbegin(), _edges.cend(), e);
  133:         bool edgeFound(ip != _edges.cend());     //  edge found
  134:         if (edgeFound) {
  135:             _edges.erase(ip);
  136:         }
  137:     }
  138:     DetermineNumberVertices(); // updates _vertices, _maxvert: called
 only once
  139: }
  140:
```

```
    1: #pragma once
    2:
    3: #include <array>
    4: #include <iostream>
    5: #include <set>                    // graph_2
    6: #include <string>
    7: #include <vector>
    8:
    9:
   10: /**
   11:  * Directed graph class.
   12:  * A better graph class that doesn't requires a consecutive numbering
 of the vertices.
   13: */
   14: class graph {
   15:     using Edge=std::array<unsigned int,2>;              // graph_3
   16: public:
   17:     /** \brief Reads edges for graph from file.
   18:      *
   19:      *  If the file  @p file_name does not exist then the code stops
with an appropriate message.
   20:      *
   21:      *  A consecutive numbering of the vertices is required.
   22:      *
   23:      * @param[in]    file_name   name of the ASCII-file
   24:      */
   25:     [[maybe_unused]] explicit graph(const std::string &file_name);
   26:
   27:     // Rule of five
   28:     graph(graph const & org) = default;
   29:     graph(graph      && org) = default;
   30:     graph& operator=(graph const & rhs) = default;
   31:     graph& operator=(graph      && rhs) = default;
   32:     ~graph()                            = default;
   33:
   34:     /**
   35:       Determines the neighboring vertices for each node from the edge
 definition.
   36:       The node itself is not contained in the neighboring vertices.
   37:
   38:       @return       vector[nn][*] with all neighboring vertices for e
ach node
   39:     */
   40:     [[nodiscard]] std::vector<std::vector<unsigned int>> get_node2nod
es() const;
   41:
   42:     /**
   43:       @return       number of edges
   44:     */
   45:     [[nodiscard]] size_t Nedges() const
   46:     {
   47:         return _edges.size();
   48:     }
   49:
   50:     /**
   51:       @return       number of vertices
   52:     */
```

```
   53:      [[nodiscard]] size_t Nvertices() const
   54:      {
   55:          return _vertices.size();          // graph_2
   56:      }
   57:
   58:       /**
   59:        @return       largest vertex index
   60:      */
   61:      [[nodiscard]] size_t Max_vertex() const             // graph_2
   62:      {
   63:          return _maxvert;
   64:      }
   65:
   66:      /** \brief Appends one directed edge to the graph.
   67:       *    The method add only edges that not already contained in the
 graph.
   68:       *
   69:       * @param[in]    v1   start vertex
   70:       * @param[in]    v2   end vertex
   71:       */
   72:      [[maybe_unused]] bool Append(unsigned int v1, unsigned int v2);
   // graph_3
   73:
   74:       /** \brief Removes one directed edge (@p v1, @p v2) from the gra
ph.
   75:       *    The method add only edges that not already contained in the
 graph.
   76:       *
   77:       * @param[in]    v1   start vertex
   78:       * @param[in]    v2   end vertex
   79:       * @return True if edge @p e exists in the graph.
   80:       */
   81:       [[maybe_unused]] bool Delete(unsigned int v1, unsigned int v2);
    // graph_3
   82:
   83:       /** \brief Removes edge @p e from the graph.
   84:       *    The method add only edges that not already contained in the
 graph.
   85:       *
   86:       * @param[in]    e   edge
   87:       * @return True if edge @p e exists in the graph.
   88:       */
   89:      bool Delete(Edge const &e);                          // graph_3
   90:
   91:       /** \brief Removes the given edges from the graph.
   92:       *    The method add only edges that not already contained in the
 graph.
   93:       *
   94:       * @param[in]    v   vector[ne],[2] of edges
   95:       * @warning No message if an edge (@p v[k][1], @p v[k][2]) doesn'
t exist in the graph.
   96:       */
   97:      void Delete(std::vector<Edge> const &v);            // graph_3
   98:
   99:       /** \brief Prints edges and vertices of the graph
  100:       *
  101:       * @param[in,out] s    output stream
```

```
  102:       * @param[in]    rhs   graph
  103:       * @return Output stream.
  104:       */
  105:      friend std::ostream& operator<<(std::ostream &s, graph const &rhs
);
  106:
  107: private:
  108:      /**
  109:        Determines the number of vertices from the edge information.
  110:        No consecutive numbering of the vertices required.
  111:      */
  112:      void DetermineNumberVertices();
  113:
  114:      std::vector<Edge>          _edges;   /**< stores the two vertices
for each edge */
  115:      std::set<unsigned int>     _vertices;/**< stores the vertex indice
s */
  116:      unsigned int               _maxvert; /**< maximal vertex index */
  117:
  118: };
```

```cpp
//graph
#include "graph.h"
#include <array>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    cout << "Hello Graph!" << endl;
    graph g1{"g_2.txt"};

    cout << g1 << endl;

    // construct mapping nodes to nodes
    auto n2n=g1.get_node2nodes();

    cout << "\n --  Nodes to Node  --\n";
    for (size_t k=0; k<n2n.size(); ++k)
    {
        cout << k << " : ";
        for (unsigned int j : n2n[k])
        {
            cout << j << " ";
        }
        cout << endl;
    }

    // ---------------
    vector<std::array<unsigned int,2>> ve{ {1,2}, {5,4}, {2,5} };
    g1.Delete(ve);
    cout << g1 << endl;

    return 0;
}
```