```
/*
   Name:     graph_scanning.h
   Author:   Leo Liberti
   Purpose:  graph scanning algorithms
   Source:   C++ header file
   History:  110123
*/

#include <iostream>
#include <string>
#include <fstream>
#include <set>
#include <map>

/**
 * The Graph class stores node IDs in a set (normally consecutive
 * ints numbered from 0), a list of arcs in a set of pairs, the
 * adjacency lists in a map from int to sets of ints, and the arc
 * costs in a map from pairs to doubles
 */
class Graph {

  //! the set of node IDs (normally 0, ..., n-1)
  std::set<int> node;

  //! the set of arcs
  std::set<std::pair<int, int> > arc;

  //! the adjacency lists
  std::map<int, std::set<int> > adj;

  //! the arc costs
  std::map<std::pair<int, int>, double> arc_cost;

 public:
  /**
   * the default class constructor
   */
  Graph();

  /**
   * the class constructor reading a graph from a file in .gph format
   * @param f is an input file stream (already opened and set on a
   * valid file with reading permissions set)
   * @see fromFile()
   */
  Graph(std::ifstream& f);

  /**
   * the default class destructor
   */
  ~Graph();

  /**
   * return the set of node IDs
   * @return a reference to the node set
   */
  std::set<int>& getNodeSet(void);

  /**
   * default method for iteratively building the graph: adds source
   * and dest to the list of nodes, adds the pair <source,dest> to the
   * list of arcs, adds the appropriate entry to the arc_cost map, and
   * updates the adjacency list (puts dest in adj[source])
   * @param source is the source node int ID
   * @param dest is the destination node int ID
   * @param cost is the arc cost
   */
  void addArc(int source, int dest, double cost);
```



```
  /**
   * parses a .gph file and reads the encoded graph into memory.
   * Format for the .gph file:
   * comments start with #, one line must contain the keyword
   * "Directed" or "Undirected" (in the latter case all edges are
   * interpreted as antiparallel pairs of arcs), and the arc lines
   * have the form "source destination cost", where source and
   * destination are integer node IDs, and cost is a double.
   * @param f is an input file stream (already opened and set on a
   * valid file with reading permissions set)
   */
  bool fromFile(std::ifstream& f);

  /**
   * print the adjacency lists of the graph
   */
  void print(void);

  /**
   * performs a Depth First Search exploration of the graph
   * @param source is the node from which the DFS search starts
   * @param visited keeps track of visited nodes (at the beginning it must
   * map every vertex to false)
   * @param spaces is an internal variable and should be set to the
   * empty string ""
   * @param method can be 1,2,3,4 according as to whether the output
   * should be a node preorder, an arc preorder, an arc postorder,
   * and a node postorder
   */
  void dfs(int source, std::map<int, bool>& visited,
           std::string& spaces, int method);
};
```

```
/*
   Name:     graph_scanning.cxx
   Author:   Leo Liberti
   Purpose:  graph scanning algorithms
   Source:   C++ implementation
   History:  110123
*/

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <list>
#include <set>
#include <vector>
#include <string>
#include <fstream>
#include "graph_scanning.h"

#define BUFSIZE 1024
#define INVALID -11111111

/**
 * the default class constructor
 */
Graph::Graph() { }

Graph::Graph(std::ifstream& f) {
  fromFile(f);
}

Graph::~Graph() { }

std::set<int>& Graph::getNodeSet(void) {
  return node;
}

void Graph::addArc(int source, int dest, double cost) {
  using namespace std;
  pair<int, int> theArc(source, dest);
  arc.insert(theArc);
  arc_cost[theArc] = cost;
  adj[source].insert(dest);
  node.insert(source);
  node.insert(dest);
}

void Graph::dfs(int v, std::map<int,bool>& visited,
                std::string& sp, int method) {
  using namespace std;
  if (method == 1) {
    cout << sp << v << endl;
  }
  set<int>::iterator sit = adj[v].begin();
  while(sit != adj[v].end()) {
    if (!visited[*sit]) {
      visited[*sit] = true;
      string spaces = sp + " ";
      if (method == 2) {
        cout << sp << "(" << v << "," << *sit << ")" << endl;
      }
      dfs(*sit, visited, spaces, method);
      if (method == 3) {
        cout << sp << "(" << v << "," << *sit << ")" << endl;
      }
    }
    sit++;
  }
  if (method == 4) {
    cout << sp << v << endl;
```



```
  }
}

bool Graph::fromFile(std::ifstream& inFile) {
  using namespace std;
  bool ret = true;

  // read in .gph arc list
  char buf[BUFSIZE];
  char* bufptr;
  char* saveptr;
  bool directed = false;
  while(!inFile.eof()) {
    // read line and find first non-blank
    inFile.getline(buf, BUFSIZE);
    bufptr = buf;
    while(*bufptr == ' ' || *bufptr == '\t') {
      bufptr++;
    }
    // if line is not a comment, parse
    if (*bufptr != '#') {
      if (strstr(buf, "Directed")) {
        directed = true;
      } else {
        int source = -1;
        int dest = -1;
        double cost = INVALID;
        // tokenize line and read three values
        char* token = strtok_r(bufptr, " \t", &saveptr);
        if (token) {
          source = atoi(token);
          token = strtok_r(NULL, " \t",  &saveptr);
          if (token) {
            dest = atoi(token);
            token = strtok_r(NULL, " \t",  &saveptr);
            if (token) {
              cost = atoi(token);
            }
          }
        }
        if (source != -1 && dest != -1 && cost != INVALID) {
          // add arc
          addArc(source, dest, cost);
        }
      }
    }
  }
  return ret;
}

void Graph::print(void) {
  using namespace std;
  pair<int,int> a;
  map<int, set<int> >::iterator mit = adj.begin();
  while(mit != adj.end()) {
    set<int>::iterator sit = mit->second.begin();
    cout << mit->first << ":";
    a.first = mit->first;
    while(sit != mit->second.end()) {
      a.second = *sit;
      cout << "[" << *sit << "," << arc_cost[a] << "]";
      sit++;
    }
    cout << endl;
    mit++;
  }
}
```

```
/*
   Name:     graph_scanning.cxx
   Author:   Leo Liberti
   Purpose:  graph scanning algorithms: main
   Source:   C++ implementation
   History:  110124
*/

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <list>
#include <set>
#include <vector>
#include <string>
#include <fstream>
#include "graph_scanning.h"

template<class T, class U> void initialize(std::map<T,U>& m,
                                           std::set<T>& k, U v) {
  using namespace std;
  m.erase(m.begin(), m.end());
  typename set<T>::iterator sit = k.begin();
  while(sit != k.end()) {
    m[*sit] = v;
    sit++;
  }
}

int main(int argc, char** argv) {

  using namespace std;

  if (argc < 4) {
    cerr << argv[0] << ": error: syntax is " << argv[0]
         << " cmd file node [arg]" << endl;
    cerr << "  cmd in {print, dfs, scan}" << endl;
    cerr << "  file describes a directed graph in .gph format" << endl;
    cerr << "  node is the source node (integer in [0, ..., n-1])" << endl;
    cerr << "  arg is traversal order for graph scanning:" << endl;
    cerr << "    1=prenode, 2=prearc, 3=postarc, 4=postnode" << endl;
    exit(1);
  }

  // parse cmd line args
  string cmd = argv[1];
  ifstream inFile(argv[2]);
  int node = atoi(argv[3]);
  int method = 0;
  if (argc == 5) {
    method = atoi(argv[4]);
  }

  Graph g;
  g.fromFile(inFile);
  inFile.close();

  string spaces = "";
  map<int,bool> visited;
  initialize<int,bool>(visited, g.getNodeSet(), false);
  if (cmd == "print") {
    g.print();
  } else if (cmd == "dfs") {
    g.dfs(node, visited, spaces, 1);
  } else if (cmd == "scan") {
    g.dfs(node, visited, spaces, method);
  }

}
```

```
CXX = c++
RM = rm

all: dfs test

dfs: graph_scanning.o dfs.cxx graph_scanning.h
	$(CXX) -o dfs dfs.cxx graph_scanning.o

graph_scanning.o: graph_scanning.cxx graph_scanning.h
	$(CXX) -c -o graph_scanning.o graph_scanning.cxx

clean:
	$(RM) -f graph_scanning.o dfs

distclean: clean
	$(RM) -f *~

test: dfs
	./dfs print test.gph 0
	./dfs dfs test.gph 0
	./dfs scan test.gph 0 4
```

```
PROJECT_NAME           = Graph Scanning
OUTPUT_DIRECTORY       = doc
EXTRACT_ALL            = YES
INPUT                  = ./
SOURCE_BROWSER         = YES
INLINE_SOURCES         = YES
STRIP_CODE_COMMENTS    = NO
HAVE_DOT               = YES
CALL_GRAPH             = YES
```