h375_graph-of-thrones.h

```
#pragma once

namespace h375
{
	void test();
	bool isBalanced(char const *_filename);
}
```

h375_graph-of-thrones.cpp

```
#include "h375_graph-of-thrones.h"

#include <iostream>		// debug
#include <cstdio>		// file io

#include <string>		// strings
#include <vector>

void h375::test()
{
	bool result = h375::isBalanced("h375_input.txt");

	// Print result
	if (result)
	{
		std::cout << "balanced\n";
	}
	else
	{
		std::cout << "not balanced\n";
	}
}

// This is a BRUTE FORCE implementation using a 2D vector as a lookup table
// Constructs a graph based on data read from a file, then
// checks all possible permutations until one of two things happens:
// 1. All permutations have been checked and they all work (true)
// 2. A permutation does not meet the conditions of local stability (false)

bool h375::isBalanced(char const* _filename)
{
	// FIRST, Open file for reading and check for validity
	FILE* myFile = fopen(_filename, "r");
	if (myFile == nullptr)
	{
		std::cout << "ERROR: NO FILE FOUND" << std::endl;
		return false;
	}

	// Prepare to read
	char myBuffer[100];

	// First line - nodes, edges
	fgets(myBuffer, 100, myFile);
	unsigned nodes = atoi(myBuffer);				// Get Nodes

	unsigned i = 0;
	while (myBuffer[i] != ' ') { ++i; }
	unsigned edges = atoi(myBuffer + i + 1);		// Get Edges

	// Read lines
	std::vector<std::string> lines;
	for (i = 0; i < edges; ++i)
	{
		fgets(myBuffer, 100, myFile);
		lines.push_back(myBuffer);
	}

	// Close file
	fclose(myFile);
```

```
// SECOND, construct the graph
std::vector<std::string> names;
std::vector<std::vector<bool>> relationships(nodes, std::vector<bool>(nodes, false));

typedef std::string::const_iterator citString;

citString myIter;                       // Walking strings to find names
bool isFriend;                          // Relationship data
std::string myFirstName;        // Holds current name
std::string mySecondName;
int indexFirstName;                     // Index of first name in names
int indexSecondName;            // Index of second name in names

// Each line adds new relationship to the graph
for (i = 0; i < edges; ++i)
{
        // Get first name
        myIter = lines[i].cbegin();
        while ((*myIter) != '+' && (*myIter) != '-') { ++myIter; }
        myFirstName = std::string(lines[i].cbegin(), myIter);
        myFirstName.pop_back();         // Empty char

        isFriend = ((*myIter) == '+');

        // Get second name
        while ((*myIter) != ' ') { ++myIter; }
        ++myIter;       // Start of actual name
        mySecondName = std::string(myIter, lines[i].cend());
        if (mySecondName.back() == '\n') { mySecondName.pop_back(); }       // newline

        indexFirstName = -1;
        indexSecondName = -1;

        // Find indices of first and second names
        for (unsigned j = 0; j < names.size(); ++j)
        {
                // Check for matching names
                if (indexFirstName < 0 && myFirstName == names[j])
                {
                        indexFirstName = j;
                }
                if (indexSecondName < 0 && mySecondName == names[j])
                {
                        indexSecondName = j;
                }

                // Can we break?
                if (indexFirstName >= 0 && indexSecondName >= 0)
                {
                        break;
                }
        }

        // Name doesn't exist within names vector - add it
        if (indexFirstName < 0)
        {
                names.push_back(myFirstName);
                indexFirstName = (names.size()-1);
        }
        if (indexSecondName < 0)
        {
                names.push_back(mySecondName);
                indexSecondName = (names.size() - 1);
        }

        // Set relationship in graph
        relationships[indexFirstName][indexSecondName] = isFriend;
        relationships[indexSecondName][indexFirstName] = isFriend;
}

// THIRD, Check graph for local stability
// How I will do this is by checking the relationship of each pair
// IF FRIENDS: they must share the SAME relationship with every other character
// IF ENEMIES: they must have OPPOSITE relationships with every other character
// When we finish checking every pair to make sure they are locally stable, return ture
// If any pairing breaks either of these criteria, return false immediately
for (unsigned a = 0; a < nodes; ++a)                    // first person
{
        for (unsigned b = a + 1; b < nodes; ++b)                // second person
```

```
		{
			for (unsigned c = b + 1; c < nodes; ++c)	// subject
			{
				if (relationships[a][b])	// FRIENDS
				{
					if (relationships[a][c] != relationships[b][c])
					{
						return false;
					}
				}
				else						// NOT FRIENDS
				{
					if (relationships[a][c] == relationships[b][c])
					{
						return false;
					}
				}
			}
		}
	}

	return true;
}
```