```cpp
// cmpt130cities.cpp : Defines the entry point for the console application.
//
#include <cmath>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

const double EARTH_RADIUS = 6371; //kms
const double PI = 3.14159;

// City Structure
struct City {
	string name;
	double latitude;
	double longitude;
};

string getFileName();
vector<City> readCities(string fname);
void printCities(const vector<City>& cities);
double distance(City c1, City c2);
double sumDistances(City c, const vector<City>& cities);
City hubCity(const vector<City>& cities);

int main()
{
	// Get file name
	string fname = getFileName();
	vector<City> cities;
	if (fname == "") {
		cout << "A valid file name was not entered" << endl;
	}
	// Process city file if possible
	else {
		cities = readCities(fname);
		if (cities.size() == 0) {

	cout << "The city file contained errors and was not processed" << endl;
		}
		else {
			printCities(cities);
			City hub = hubCity(cities);
			cout << endl << "The hub city is " << hub.name << endl << endl;
		}
	}

	return 0;
}

// POST: Returns a (city) file name entered by the user or
//       the empty string if the user does not enter a valid file name
```

```
string getFileName()
{
	string result = "";

	cout << "Enter a city file name: ";
	cin >> result;

	ifstream fileIn(result);

	// Repeatedly request user input if file cannot be opened
	while (fileIn.fail() && result != "q") {
		fileIn.close();
		cout << "Enter a city file name or 'q' to quit: ";
		cin >> result;
		fileIn.open(result);
	}

	// If user choose to quit set result to empty string
	if (result == "q") {
		result = "";
	}

	fileIn.close();
	return result;
}//getFileName

// PARAM: fname - name of the city file to be opened
// POST: Returns a vector of cities stored in the file named in
//       the parameter, returns an empty vector if the file contains
//       *any* lines that cannot be processed as a city
vector<City> readCities(string fname)
{
	vector<City> result;
	bool badFile = false;

	ifstream fileIn(fname);

	// Process lines until end of file reached or a line cannot be processed
	while (!(fileIn.eof()) && !badFile) {
		City tempCity;
		if (fileIn >> tempCity.name >> tempCity.latitude >> tempCity.longitude) {
			result.push_back(tempCity);
		}
		// Empty vector if file contains errors
		else {
			badFile = true;
			result.clear();
		}
	}

	fileIn.close();
	return result;
}//readCities

// PARAM: cities - vector to be printed
```

```
// POST: Prints the city data in the input vector, one city per line
void printCities(const vector<City>& cities)
{
        // Determine maximum length of a city name
        int nameWidth = 0;
        for (City c : cities) {
                if (c.name.size() + 1 > nameWidth) {
                        nameWidth = c.name.size() + 1;
                }
        }

        // Set variables for printing latitude and longitude
        int latLongWidth = 10;
        int precision = 2;

        // Print Titles
        cout << setw(nameWidth) << "CITY";
        cout << setw(latLongWidth) << "LATITUDE";
        cout << setw(latLongWidth) << "LONGITUDE" << endl;

        // Print dashes under titles
        cout << setfill('-');
        cout << setw(nameWidth+1) << " ";
        cout << setw(latLongWidth) << " ";
        cout << setw(latLongWidth) << " " << endl;
        cout << setfill(' ');

        // Print city data
        for (City c : cities) {
                cout << fixed << setprecision(precision);
                cout << setw(nameWidth) << c.name;
                cout << setw(latLongWidth) << c.latitude;
                cout << setw(latLongWidth) << c.longitude << endl;
        }
}//printCities

// PARAM c1 and c2 - cities to compute distance between
// POST: Returns the great circle distance between c1 and c2
double distance(City c1, City c2)
{
        // Convert degrees to radians
        double lat1 = c1.latitude * PI / 180;
        double long1 = c1.longitude * PI / 180;
        double lat2 = c2.latitude * PI / 180;
        double long2 = c2.longitude * PI / 180;

        return acos(sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(long2-
long1)) * EARTH_RADIUS;
}//distance

// PARAM: c - city to compute distance from
//        cities - list of cities to sum distances to
// POST: Returns the sum of the distances between c and all cities in cities vector
double sumDistances(City c, const vector<City>& cities)
{
```

```
        double result = 0;

        for (City destination : cities) {
                result += distance(c, destination);
        }

        return result;
}//sumDistances

// PRE: cities is not empty
// PARAM: cities - vector of cities
// POST: Returns the city with the least sum distance between it and all other
//       cities in the cities vector - i.e. the hub city
City hubCity(const vector<City>& cities)
{
        City result = cities[0];
        double minDistance = sumDistances(cities[0], cities);

        for (City nextCity : cities) {
                double nextDistance = sumDistances(nextCity, cities);
                if (nextDistance < minDistance) {
                        result = nextCity;
                        minDistance = nextDistance;
                }
        }

        return result;
}//hubCity
```