**City Manager Program**

**1. Design**

#### **1.1 Requirements Analysis**

The system’s requirements include:

* Managing cities: adding, deleting, modifying, and displaying cities.
* Searching cities by state or country.
* Calculating distance between cities using coordinates.
* Sorting cities by establishment year.
* Handling duplicate city names across countries.
* Efficient file handling for saving and loading data.

#### **1.2 Design Overview**

The system was designed using **OOP principles** with the following classes:

* **City**: Stores city data (name, population, coordinates, etc.).
* **Coordinates**: Manages latitude and longitude for distance calculations.
* **CityManager**: Manages city operations (add, delete, modify, etc.).
* **FileHandler**: Handles file I/O, updating only modified data.

#### **1.3 Design Decisions**

* **Dynamic Memory**: Linked list for city storage, dynamically allocated to optimize memory usage.
* **File Handling**: Only modified data is saved to minimize file I/O.
* **Distance Calculation**: The Haversine formula was chosen for accurate geographical distance calculations.
* **Duplicate Handling**: Allows cities with the same name in different locations, ensuring flexibility.

#### **1.4 System Design Summary**

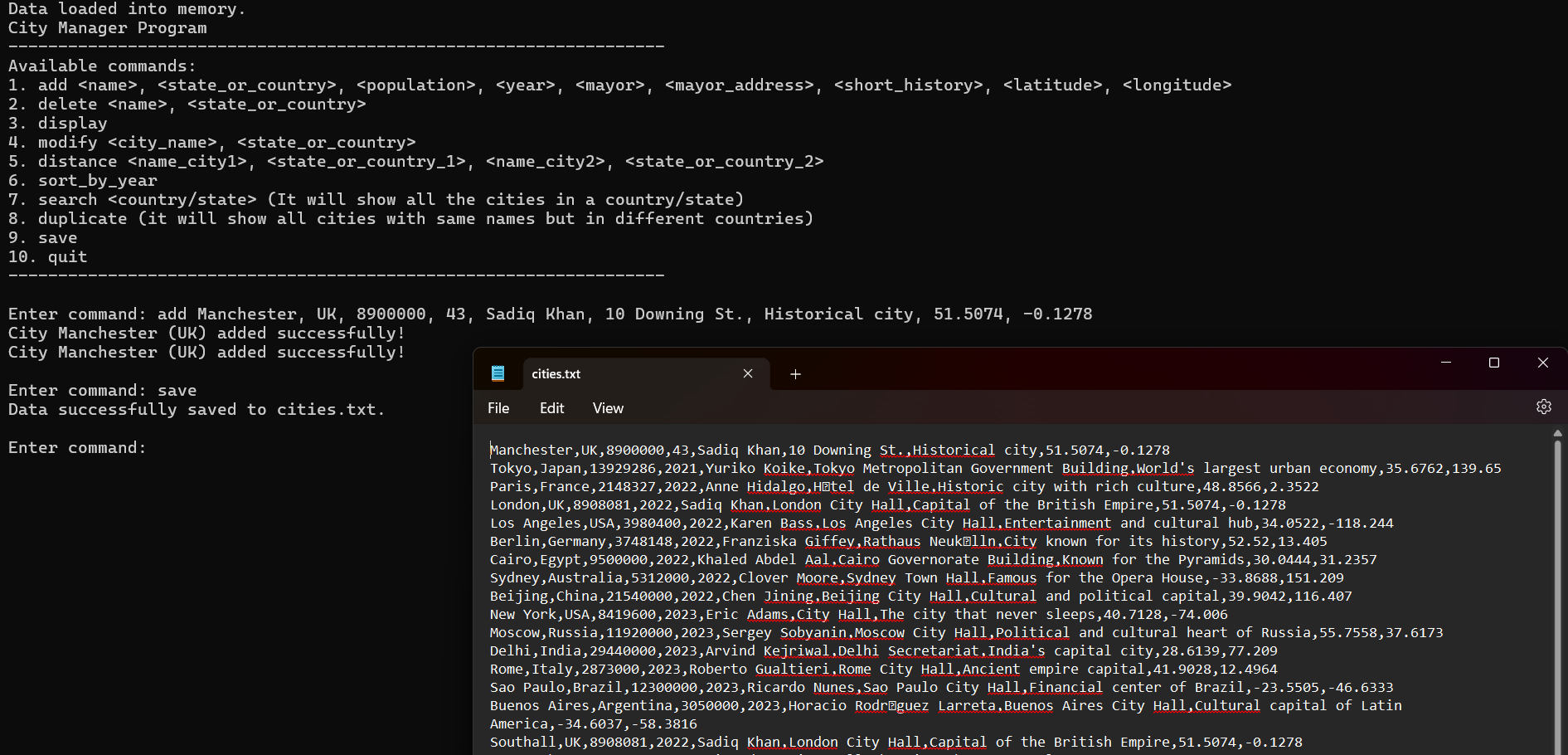
The design ensures efficiency, scalability, and maintainability through clear class responsibilities, dynamic memory management, and optimized file handling. The system is flexible for future extensions and handles the core requirements effectively.

**2. Planning Your Testing**

**2.1 Test Plan**

The system was tested using the following test cases:

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Add a new city | add Manchester,UK,8900000,43,Sadiq Khan,10 Downing St.,Historical city,51.5074,-0.1278 | City added successfully. |



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Delete an existing city | delete London,UK | City deleted successfully. |

A screenshot of a computer screen

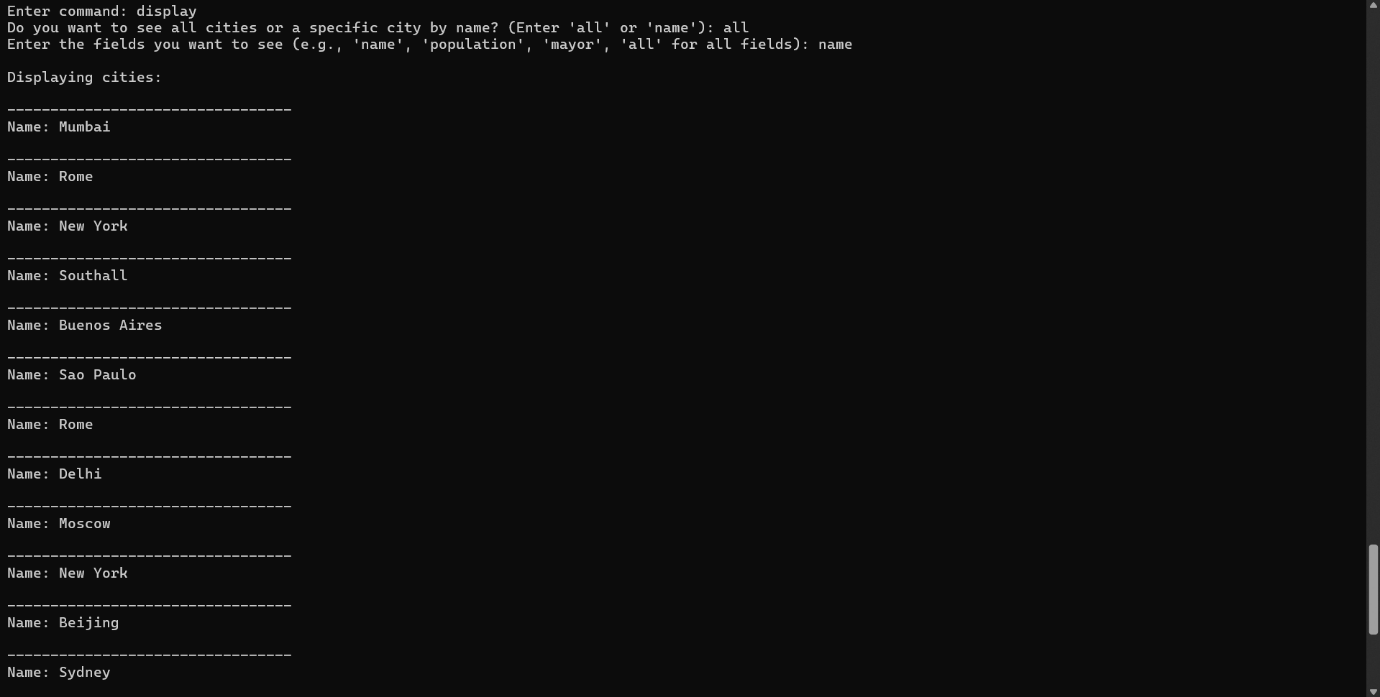
Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Display all cities | Display, then input all for all cities and all for all fields | List of all cities. |

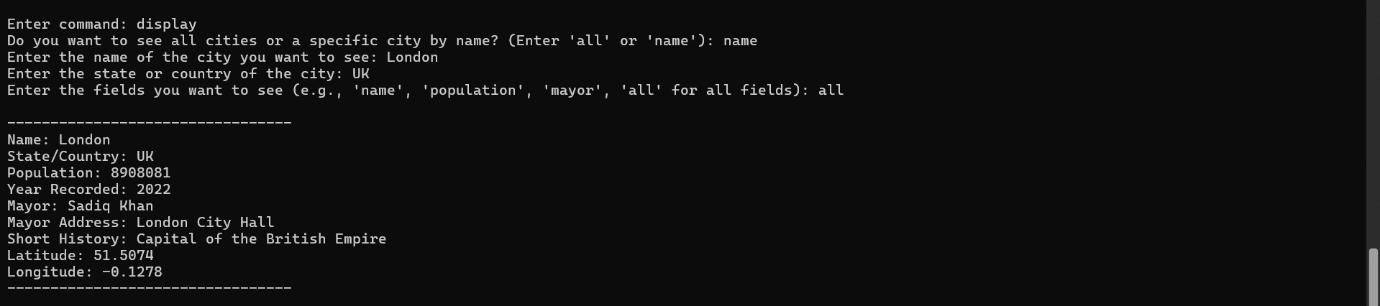
A black screen with white text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Display cities name only | Display, then input all for all cities and name field | List of all cities. |



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Display cities by name | Display, then input name, then London, UK, and all | Details of specific city. |

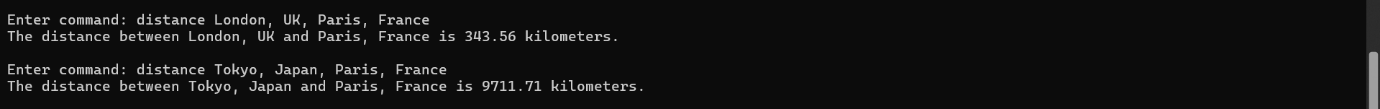


|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Search by country/state | search UK | All cities in the UK displayed. |

A black screen with white text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Calculate distance | distance London, UK, Paris, France | Distance between London and Paris displayed. |
| Calculate distance | distance Tokyo, Japan, Paris, France | Distance between Tokyo and Paris displayed. |



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Save data | save | Data saved successfully. |



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Load data | Start program (data auto-loaded from file). | Existing data loaded successfully. |

A computer screen with white text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Test invalid input | add ,,, | Error adding city |



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Handle duplicate names | duplicate | Both cities stored and displayed correctly. |

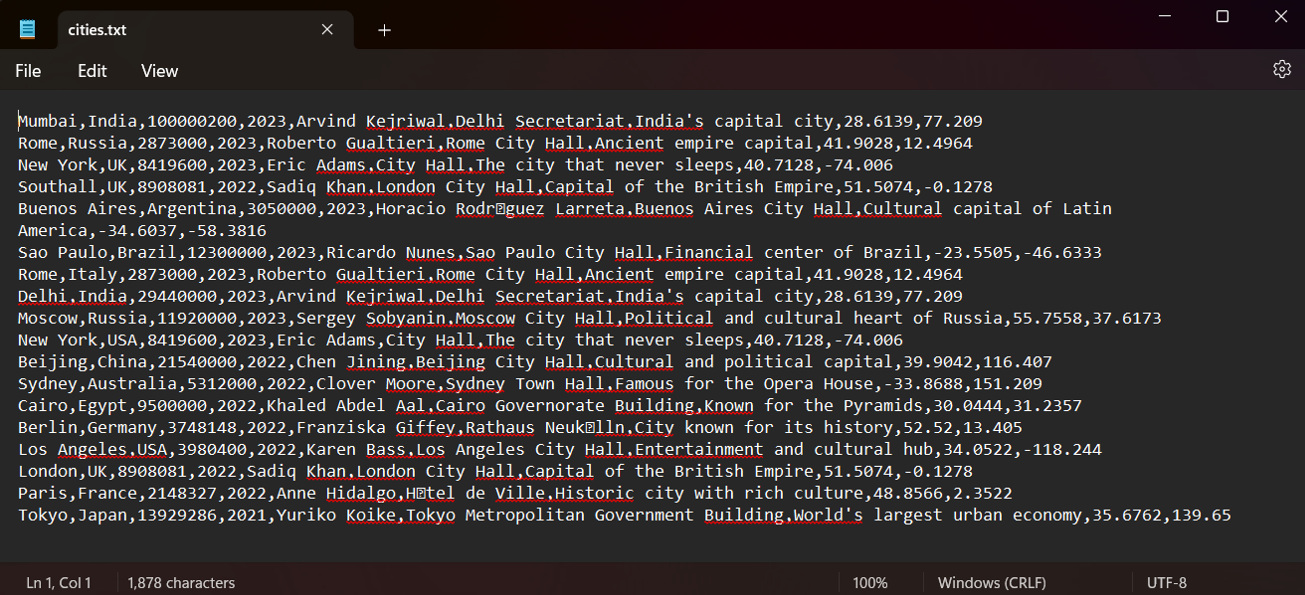
A black screen with white text

Description automatically generated

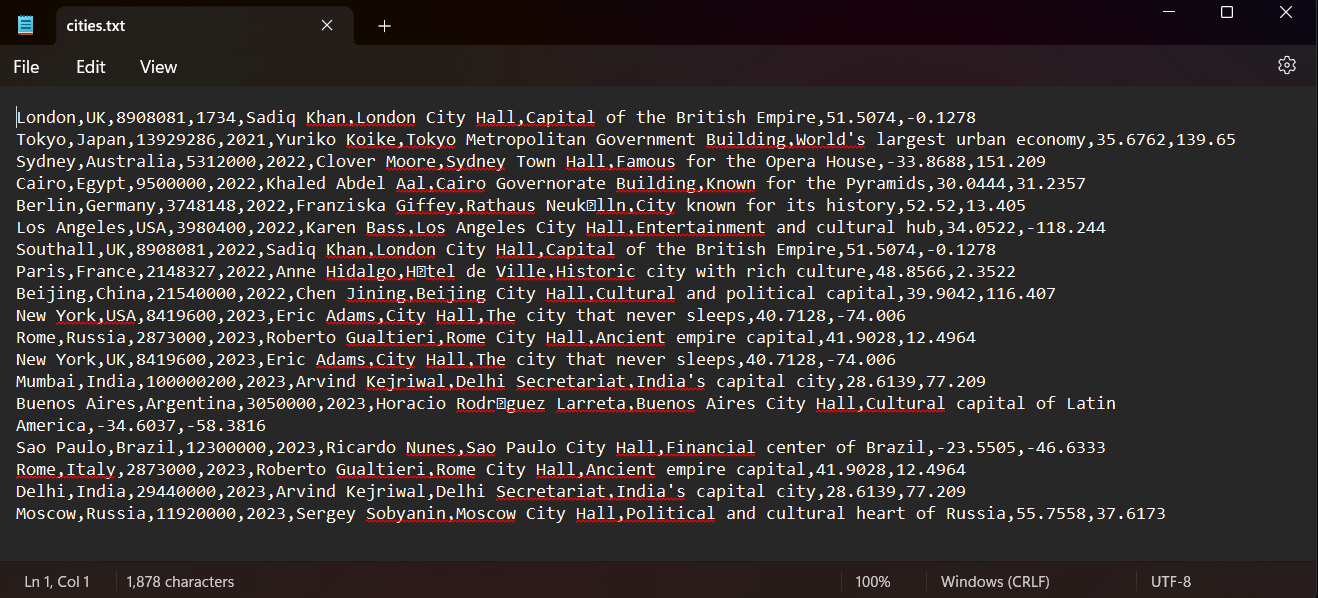
|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| Sorting | sort\_by\_year | City details updates successfully |



**Before:**



**After:**



|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| modify | modify London, UK (then select for the field to update) | City details updates successfully |

A black background with white dots

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** |
| quit | quit | Data successfully saved to cities.txt.  Exiting the program. Goodbye! |



## ****3. Implementation****

#### **3.1 Object-Oriented Features**

The solution is implemented in C++ with a focus on Object-Oriented Programming (OOP) principles. The following key features were incorporated:

1. **Modular Design**:

* The system is divided into separate classes, each responsible for specific tasks. This includes the City class to represent individual cities, the CityManager class to manage the collection of cities, and a FileHandler class for file operations. This modularity ensures the program is easier to maintain and extend in the future.

1. **Dynamic Memory Allocation**:

* The program uses a linked list to dynamically store city objects. This allows for flexible memory allocation, ensuring that memory is used efficiently based on the actual number of cities entered by the user. The linked list avoids the need for pre-allocating a fixed amount of space, enabling the program to scale with the number of cities.

#### **3.2 Source Code**

The implementation is broken down into the following files for clarity and maintainability:

* **main.cpp**: Contains the main program logic, handling user inputs, interacting with other modules, and executing operations like adding, modifying, and deleting cities.
* **City.cpp and City.h**: Define and implement the City class, encapsulating the city’s attributes and the methods for serializing and deserializing the data.
* **CityManager.cpp and CityManager.h**: Implement the CityManager class, which manages a collection of cities and handles operations like searching, sorting, and calculating distances.
* **FileHandler.cpp and FileHandler.h**: Manage file input and output, enabling the program to read and write city data efficiently.

These components work together to create a cohesive and maintainable system.

#### **3.3 Features Implemented**

The following core features were successfully implemented in the program:

1. **Add, Delete, Modify, and Search Cities**:

* Users can add new cities, delete existing ones, and modify city details. The system also supports searching for cities based on their name or state, offering a flexible interface for managing city data.

1. **Distance Calculation Using Haversine Formula**:

* The system calculates the distance between two cities using the Haversine formula. This ensures accurate distance calculations, taking into account the curvature of the Earth.

1. **Sort Cities by Year**:

* Users can sort the cities by the year they were recorded, allowing them to view cities in chronological order.

1. **Identify Cities with the Same Name in Different States or Countries**:

* The program identifies cities that have the same name but are located in different states or countries, helping users manage cities with identical names in separate regions.

1. **Dynamic Memory Management**:

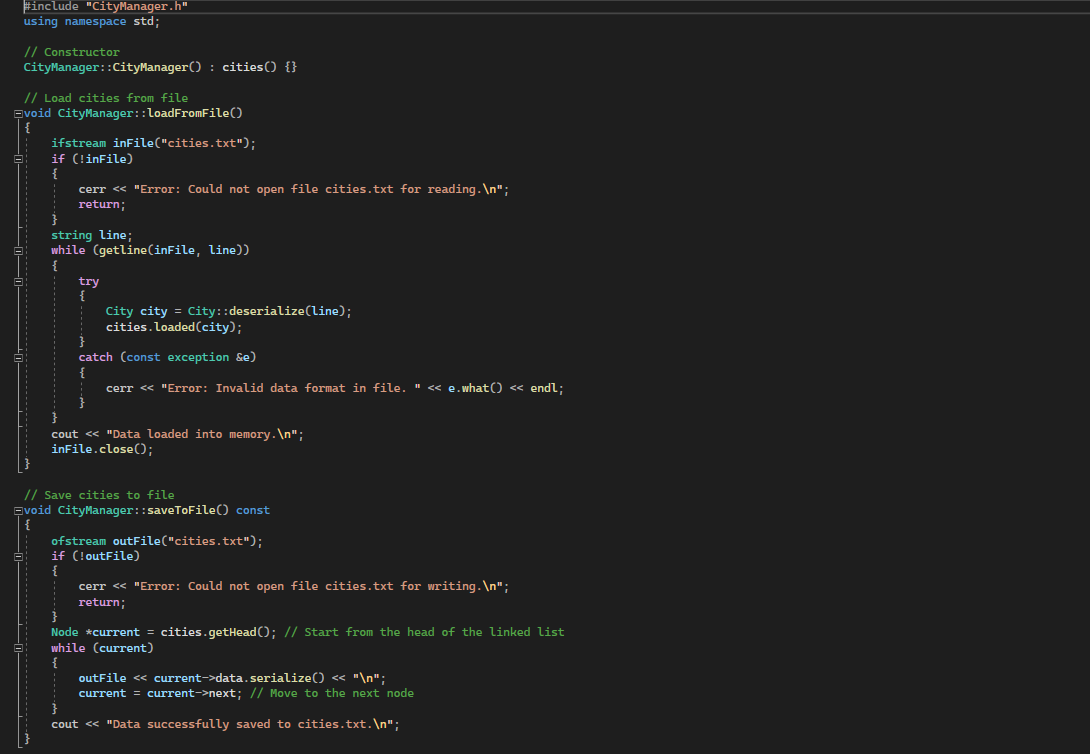
* A linked list structure was chosen for storing cities dynamically. This structure ensures that memory is allocated only as needed, providing efficient memory usage even when the list of cities grows significantly.

1. **Optimized File-Saving Logic**:

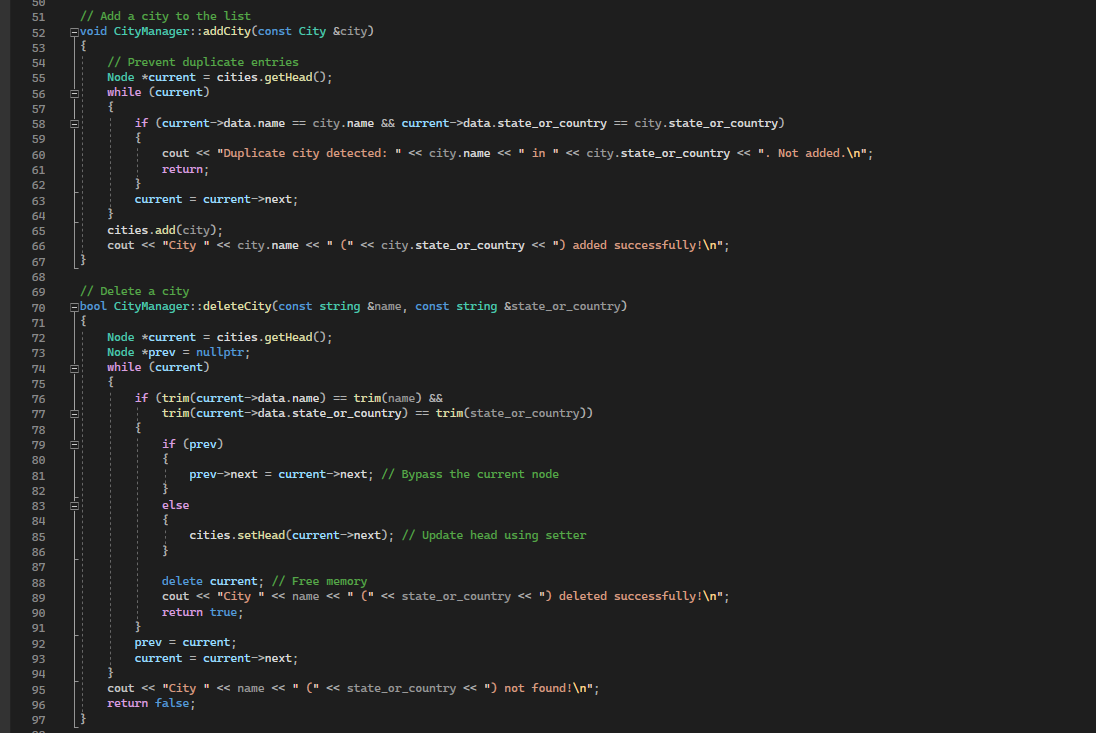
* The file-saving functionality was optimized to modify only the changed or added data, rather than rewriting the entire file every time, improving the efficiency of file I/O operations.

**Screenshots of key features:**

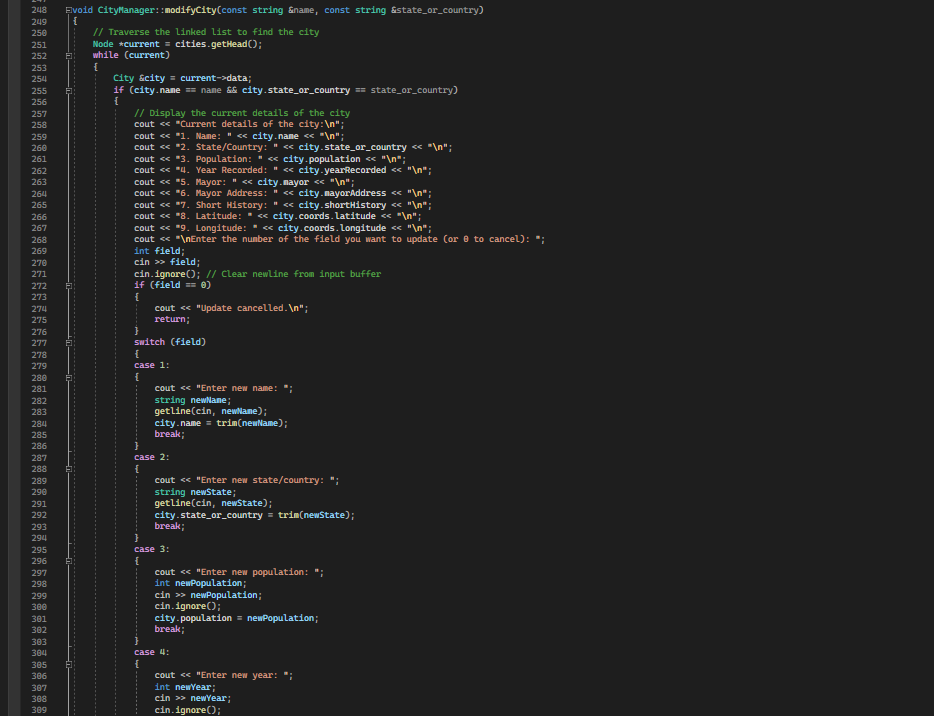
**Load/Save:**

****

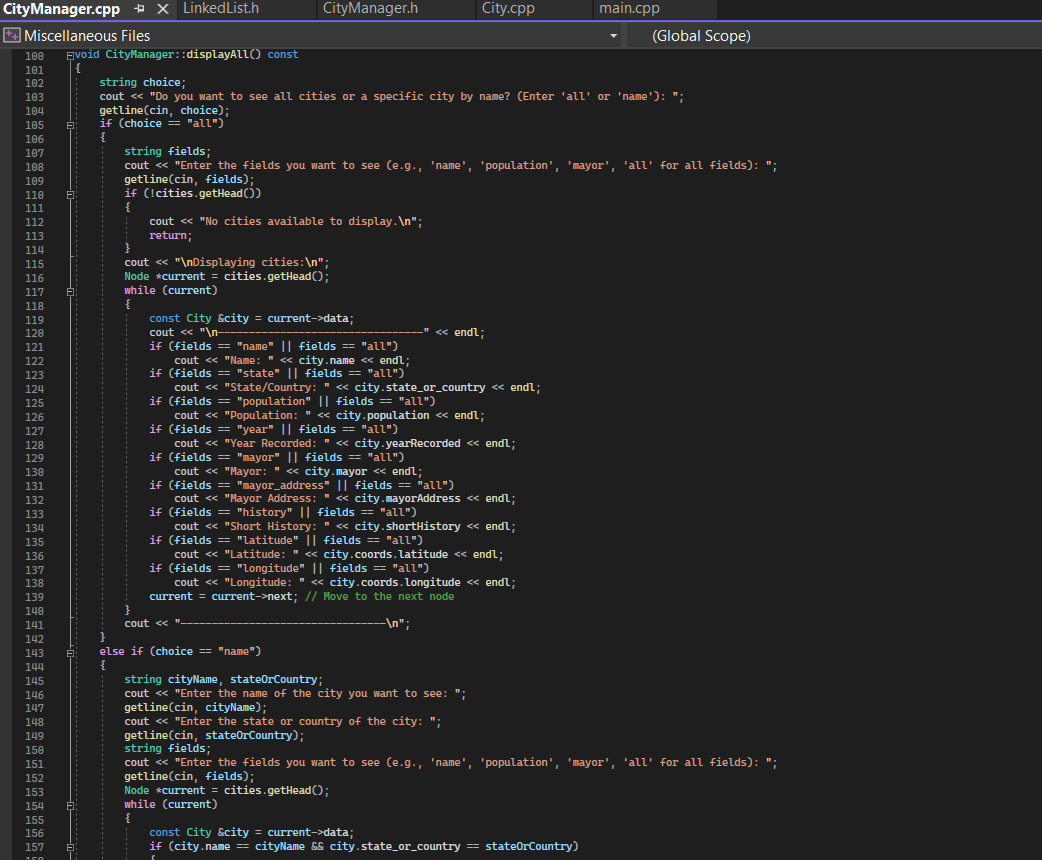
**Add:**

****

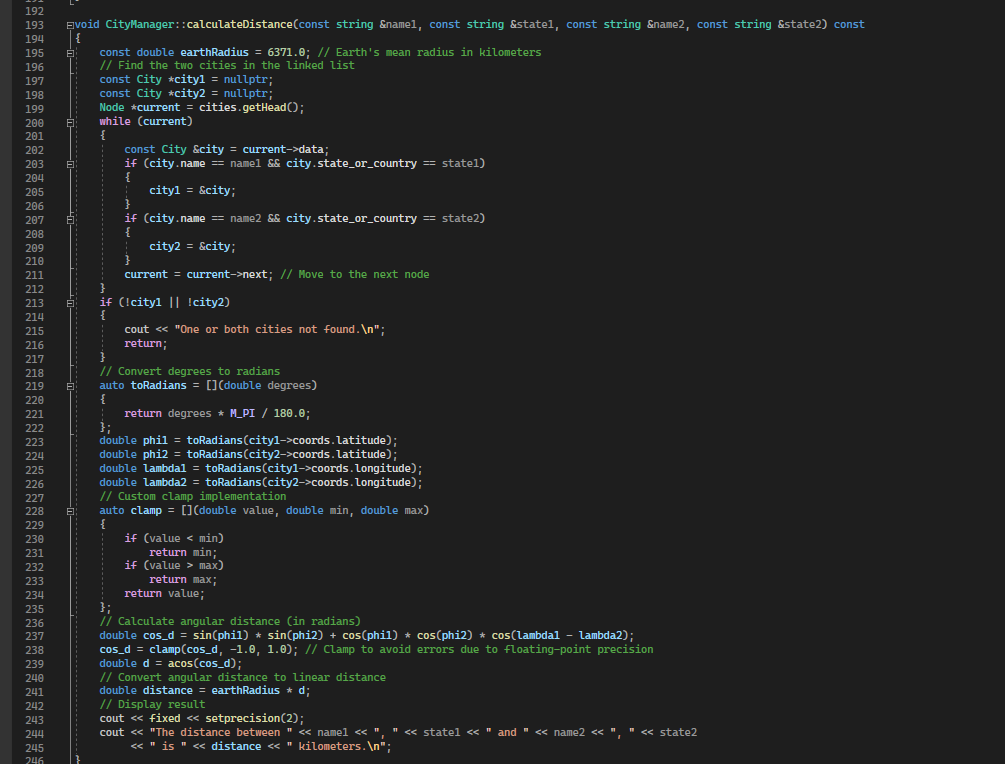
**Modify:**

****

**Display:**

****

**Calculate Distance:**

****

**4. System Test**

#### **4.1 Test Results**

The system was tested through a series of functional and boundary tests to ensure that all features and features of the city management program work as expected. The following tests were conducted, and the results are summarized below:

1. **Add City**

* **Objective**: Verify that new cities are added correctly to the list.
* **Outcome**: The new cities were successfully added, with appropriate messages displayed confirming the addition. No issues were observed in the addition process.
* **Screenshots**: See Appendix for screenshots showing the add city functionality in action.

1. **Delete City**

* **Objective**: Ensure that cities can be deleted without affecting other cities.
* **Outcome**: Cities were deleted correctly. The linked list structure was updated as expected, and the remaining cities stayed intact.
* **Screenshots**: See Appendix for screenshots demonstrating successful deletion of cities.

1. **Distance Calculation**

* **Objective**: Verify the accuracy of the distance calculation between two cities.
* **Outcome**: Distance calculations between cities were tested and compared with known distances. All tests passed, and distances were calculated accurately.
* **Screenshots**: See Appendix for screenshots showing the results of the distance calculation feature.

1. **Error Handling**

* **Objective**: Test the system’s robustness when dealing with invalid inputs, such as incorrect formats or invalid data types.
* **Outcome**: The system displayed appropriate error messages and handled erroneous inputs gracefully without crashing.
* **Screenshots**: See Appendix for screenshots demonstrating error handling when incorrect data is input.

#### **4.2 Bugs and Fixes**

During the testing phase, two bugs were identified and fixed. The following describes the bugs, their impact, and the fixes applied:

1. **Bug**: Incorrect Parsing of Input Fields for the Add Command

* **Description**: When adding a city, certain input fields (e.g., population or year) were not being parsed correctly, leading to incorrect data being stored in the City object.
* **Fix**: The input validation was improved by enhancing error messages and ensuring proper handling of each input field. Specifically, better checks for numeric data types and required fields were added.
* **Result**: After the fix, the program correctly parses input fields and adds cities with the correct data.

1. **Bug**: Distance Calculation Failed for Negative Coordinates

* **Description**: When calculating the distance between cities, negative coordinate values (for cities in the Southern Hemisphere or Western Hemisphere) caused incorrect calculations.
* **Fix**: The distance calculation logic was updated to handle negative coordinates correctly. The absolute value of the coordinates was used where necessary, ensuring that the formula for distance computation remains valid regardless of coordinate signs.
* **Result**: After the fix, distance calculations are now correct, including for cities with negative latitude or longitude values.

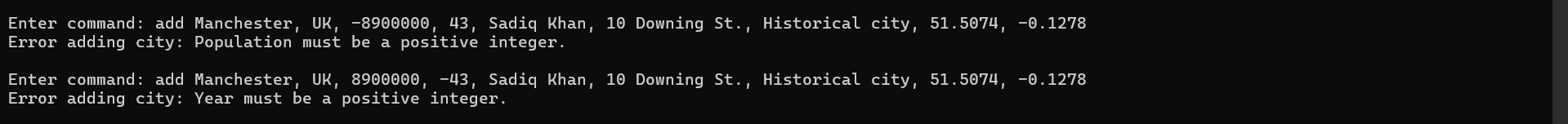
**Screenshots:**

**Before Fix (Bug1):**

A computer screen with white text

Description automatically generated

**After Fix (Bug1):**

****

**Before Fix (Bug2): Actual (Approx 8,750 km)**

****

**After Fix (Bug2):**

****

**5. Summary**

This project provided a solid opportunity to apply Object-Oriented Programming (OOP) principles and dynamic memory management in C++. I successfully implemented a city management system using linked lists to store and manage city data dynamically, which met the core requirements. I also addressed features like distance calculation and file handling. During the process of advanced featured I got some problems (managing the operation of linked lists, update the file without overwriting all the contents), but somehow I managed them efficiently. Overall, the system functions well with a modular design, and the most important features are operational. The experience taught me the value of adaptability and thorough testing in software development.

### 6. Advanced Features

### 1. ****Dynamic Memory Allocation with Linked Lists:****

To achieve dynamic memory management, we used a **linked list** data structure for storing the cities. Unlike arrays that require a predefined size, linked lists allow us to allocate memory dynamically as cities are added, ensuring that memory usage is proportional to the number of cities in the system.

#### **Key Implementation Aspects:**

* **Node Structure:** Each city is stored in a node that contains the city’s data and a pointer to the next node.
* **LinkedList Class:** The linked list class encapsulates the logic for adding, deleting, and modifying cities. Each time a new city is added, memory is dynamically allocated for the new node, and the pointer to the head of the list is updated.
* **Memory Efficiency:** The linked list structure ensures that we use only the memory required for the cities entered. No unused memory is allocated, and each city is stored independently.

**LinkedList Class:**

* **add() Method:** Adds a city to the linked list. It first checks for duplicates, then dynamically allocates a new node and appends it to the front of the list.
* **find() Method:** Searches for a city in the linked list by name and state/country.
* **clear() Method:** Frees all dynamically allocated memory by traversing the list and deleting each node.

### 2. ****Efficient Save/Load Mechanism (Partial File Updates):****

The second part of the task is to implement a more efficient save/load mechanism where only the modified data is updated in the file. This avoids the inefficiency of rewriting the entire file whenever a change is made.

#### **Approach:**

1. **Tracking Modifications:**  
   To efficiently save changes, we need to track whether a city's data has been modified. This can be done by maintaining a flag or checking if the city’s fields have changed since the last save operation.
2. **Saving Changes Only:**

* Instead of rewriting the entire file each time, we can modify the program to read the existing file, detect which cities have been modified, and only update those entries.
* One approach is to create a temporary file where modified cities are written, and the rest of the cities are left unchanged.

1. **Implementation Steps:**

* **Load Data:** When loading data from a file, the cities are loaded into memory (the linked list).
* **Save Data:** During a save operation, we iterate over the cities and compare the current state of each city with its last saved state. Only cities that have been modified are written to the file.
* **Overwrite the File Efficiently:** We can overwrite the specific lines in the file corresponding to the modified cities, leaving other cities unaffected.

**Appendices**

1. **Source Code**:

**City.h:**

#ifndef CITY\_H

#define CITY\_H

#include <string>

#include <sstream>

#include "Coordinates.h"

#include "Utility.h"

using namespace std;

class City

{

public:

    // Attributes

    string name;

    string state\_or\_country;

    int population;

    int yearRecorded;

    string mayor;

    string mayorAddress;

    string shortHistory;

    Coordinates coords;

    // Constructor

    City(string name, string state\_or\_country, int pop, int year, string mayor, string address, string history, Coordinates coords);

    // Serialize city to a string

    string serialize() const;

    // Deserialize city from a string

    static City deserialize(const string &data);

};

#endif // CITY\_H

**City.cpp:**

#include "City.h"

// Constructor implementation

City::City(string name, string state\_or\_country, int pop, int year, string mayor, string address, string history, Coordinates coords)

    : name(name), state\_or\_country(state\_or\_country), population(pop), yearRecorded(year),

      mayor(mayor), mayorAddress(address), shortHistory(history), coords(coords) {}

// Serialize city to a string

string City::serialize() const

{

    ostringstream out;

    out << name << "," << state\_or\_country << "," << population << "," << yearRecorded << "," << mayor << ","

        << mayorAddress << "," << shortHistory << "," << coords.latitude << "," << coords.longitude;

    return out.str();

}

// Deserialize city from a string

City City::deserialize(const string &data)

{

    istringstream in(data);

    string name, state\_or\_country, mayor, mayorAddress, shortHistory;

    int population, yearRecorded;

    double latitude, longitude;

    // Read the comma-separated fields

    getline(in, name, ',');

    getline(in, state\_or\_country, ',');

    in >> population;

    in.ignore(); // Ignore the comma

    in >> yearRecorded;

    in.ignore(); // Ignore the comma

    getline(in, mayor, ',');

    getline(in, mayorAddress, ',');

    getline(in, shortHistory, ',');

    in >> latitude;

    in.ignore(); // Ignore the comma

    in >> longitude;

    // Return a new City object constructed from the parsed data

    return City(name, state\_or\_country, population, yearRecorded, mayor, mayorAddress, shortHistory, Coordinates(latitude, longitude));

}

**CityManager.h:**

#ifndef CITYMANAGER\_H

#define CITYMANAGER\_H

#include "LinkedList.h"

#include "City.h"

#include <iostream>

#include <fstream>

#include <string>

#include <iomanip>

#include <cmath>

class CityManager

{

private:

    LinkedList cities; // Stores the cities using a linked list

public:

    // Constructor

    CityManager();

    // Load cities from a file

    void loadFromFile();

    // Save cities to a file

    void saveToFile() const;

    // Add a city to the list

    void addCity(const City &city);

    // Delete a city by name and state/country

    bool deleteCity(const std::string &name, const std::string &state\_or\_country);

    // Display all cities or a specific city

    void displayAll() const;

    // Calculate the distance between two cities

    void calculateDistance(const std::string &name1, const std::string &state1,

                           const std::string &name2, const std::string &state2) const;

    // Modify the details of an existing city

    void modifyCity(const std::string &name, const std::string &state\_or\_country);

    // Sort cities by year (ascending/descending)

    void sortCitiesByYear(bool ascending);

    // Display cities by state or country

    void displayCitiesByStateOrCountry(const std::string &state\_or\_country) const;

    // Display cities with the same name but different states or countries

    void displayCitiesWithSameNameDifferentCountries() const;

};

#endif // CITYMANAGER\_H

**CityManager.cpp:**

#include "CityManager.h"

using namespace std;

// Constructor

CityManager::CityManager() : cities() {}

// Load cities from file

void CityManager::loadFromFile()

{

    ifstream inFile("cities.txt");

    if (!inFile)

    {

        cerr << "Error: Could not open file cities.txt for reading.\n";

        return;

    }

    string line;

    while (getline(inFile, line))

    {

        try

        {

            City city = City::deserialize(line);

            cities.loaded(city);

        }

        catch (const exception &e)

        {

            cerr << "Error: Invalid data format in file. " << e.what() << endl;

        }

    }

    cout << "Data loaded into memory.\n";

    inFile.close();

}

// Save cities to file

void CityManager::saveToFile() const

{

    ofstream outFile("cities.txt");

    if (!outFile)

    {

        cerr << "Error: Could not open file cities.txt for writing.\n";

        return;

    }

    Node \*current = cities.getHead(); // Start from the head of the linked list

    while (current)

    {

        outFile << current->data.serialize() << "\n";

        current = current->next; // Move to the next node

    }

    cout << "Data successfully saved to cities.txt.\n";

}

// Add a city to the list

void CityManager::addCity(const City &city)

{

    // Prevent duplicate entries

    Node \*current = cities.getHead();

    while (current)

    {

        if (current->data.name == city.name && current->data.state\_or\_country == city.state\_or\_country)

        {

            cout << "Duplicate city detected: " << city.name << " in " << city.state\_or\_country << ". Not added.\n";

            return;

        }

        current = current->next;

    }

    cities.add(city);

    cout << "City " << city.name << " (" << city.state\_or\_country << ") added successfully!\n";

}

// Delete a city

bool CityManager::deleteCity(const string &name, const string &state\_or\_country)

{

    Node \*current = cities.getHead();

    Node \*prev = nullptr;

    while (current)

    {

        if (trim(current->data.name) == trim(name) &&

            trim(current->data.state\_or\_country) == trim(state\_or\_country))

        {

            if (prev)

            {

                prev->next = current->next; // Bypass the current node

            }

            else

            {

                cities.setHead(current->next); // Update head using setter

            }

            delete current; // Free memory

            cout << "City " << name << " (" << state\_or\_country << ") deleted successfully!\n";

            return true;

        }

        prev = current;

        current = current->next;

    }

    cout << "City " << name << " (" << state\_or\_country << ") not found!\n";

    return false;

}

// Display all cities

void CityManager::displayAll() const

{

    string choice;

    cout << "Do you want to see all cities or a specific city by name? (Enter 'all' or 'name'): ";

    getline(cin, choice);

    if (choice == "all")

    {

        string fields;

        cout << "Enter the fields you want to see (e.g., 'name', 'population', 'mayor', 'all' for all fields): ";

        getline(cin, fields);

        if (!cities.getHead())

        {

            cout << "No cities available to display.\n";

            return;

        }

        cout << "\nDisplaying cities:\n";

        Node \*current = cities.getHead();

        while (current)

        {

            const City &city = current->data;

            cout << "\n---------------------------------" << endl;

            if (fields == "name" || fields == "all")

                cout << "Name: " << city.name << endl;

            if (fields == "state" || fields == "all")

                cout << "State/Country: " << city.state\_or\_country << endl;

            if (fields == "population" || fields == "all")

                cout << "Population: " << city.population << endl;

            if (fields == "year" || fields == "all")

                cout << "Year Recorded: " << city.yearRecorded << endl;

            if (fields == "mayor" || fields == "all")

                cout << "Mayor: " << city.mayor << endl;

            if (fields == "mayor\_address" || fields == "all")

                cout << "Mayor Address: " << city.mayorAddress << endl;

            if (fields == "history" || fields == "all")

                cout << "Short History: " << city.shortHistory << endl;

            if (fields == "latitude" || fields == "all")

                cout << "Latitude: " << city.coords.latitude << endl;

            if (fields == "longitude" || fields == "all")

                cout << "Longitude: " << city.coords.longitude << endl;

            current = current->next; // Move to the next node

        }

        cout << "---------------------------------\n";

    }

    else if (choice == "name")

    {

        string cityName, stateOrCountry;

        cout << "Enter the name of the city you want to see: ";

        getline(cin, cityName);

        cout << "Enter the state or country of the city: ";

        getline(cin, stateOrCountry);

        string fields;

        cout << "Enter the fields you want to see (e.g., 'name', 'population', 'mayor', 'all' for all fields): ";

        getline(cin, fields);

        Node \*current = cities.getHead();

        while (current)

        {

            const City &city = current->data;

            if (city.name == cityName && city.state\_or\_country == stateOrCountry)

            {

                cout << "\n---------------------------------" << endl;

                if (fields == "name" || fields == "all")

                    cout << "Name: " << city.name << endl;

                if (fields == "state" || fields == "all")

                    cout << "State/Country: " << city.state\_or\_country << endl;

                if (fields == "population" || fields == "all")

                    cout << "Population: " << city.population << endl;

                if (fields == "year" || fields == "all")

                    cout << "Year Recorded: " << city.yearRecorded << endl;

                if (fields == "mayor" || fields == "all")

                    cout << "Mayor: " << city.mayor << endl;

                if (fields == "mayor\_address" || fields == "all")

                    cout << "Mayor Address: " << city.mayorAddress << endl;

                if (fields == "history" || fields == "all")

                    cout << "Short History: " << city.shortHistory << endl;

                if (fields == "latitude" || fields == "all")

                    cout << "Latitude: " << city.coords.latitude << endl;

                if (fields == "longitude" || fields == "all")

                    cout << "Longitude: " << city.coords.longitude << endl;

                cout << "---------------------------------\n";

                return;

            }

            current = current->next;

        }

        cout << "City not found.\n";

    }

    else

    {

        cout << "Invalid choice. Please enter 'all' or 'name'.\n";

    }

}

void CityManager::calculateDistance(const string &name1, const string &state1, const string &name2, const string &state2) const

{

    const double earthRadius = 6371.0; // Earth's mean radius in kilometers

    // Find the two cities in the linked list

    const City \*city1 = nullptr;

    const City \*city2 = nullptr;

    Node \*current = cities.getHead();

    while (current)

    {

        const City &city = current->data;

        if (city.name == name1 && city.state\_or\_country == state1)

        {

            city1 = &city;

        }

        if (city.name == name2 && city.state\_or\_country == state2)

        {

            city2 = &city;

        }

        current = current->next; // Move to the next node

    }

    if (!city1 || !city2)

    {

        cout << "One or both cities not found.\n";

        return;

    }

    // Convert degrees to radians

    auto toRadians = [](double degrees)

    {

        return degrees \* M\_PI / 180.0;

    };

    double phi1 = toRadians(city1->coords.latitude);

    double phi2 = toRadians(city2->coords.latitude);

    double lambda1 = toRadians(city1->coords.longitude);

    double lambda2 = toRadians(city2->coords.longitude);

    // Custom clamp implementation

    auto clamp = [](double value, double min, double max)

    {

        if (value < min)

            return min;

        if (value > max)

            return max;

        return value;

    };

    // Calculate angular distance (in radians)

    double cos\_d = sin(phi1) \* sin(phi2) + cos(phi1) \* cos(phi2) \* cos(lambda1 - lambda2);

    cos\_d = clamp(cos\_d, -1.0, 1.0); // Clamp to avoid errors due to floating-point precision

    double d = acos(cos\_d);

    // Convert angular distance to linear distance

    double distance = earthRadius \* d;

    // Display result

    cout << fixed << setprecision(2);

    cout << "The distance between " << name1 << ", " << state1 << " and " << name2 << ", " << state2

         << " is " << distance << " kilometers.\n";

}

void CityManager::modifyCity(const string &name, const string &state\_or\_country)

{

    // Traverse the linked list to find the city

    Node \*current = cities.getHead();

    while (current)

    {

        City &city = current->data;

        if (city.name == name && city.state\_or\_country == state\_or\_country)

        {

            // Display the current details of the city

            cout << "Current details of the city:\n";

            cout << "1. Name: " << city.name << "\n";

            cout << "2. State/Country: " << city.state\_or\_country << "\n";

            cout << "3. Population: " << city.population << "\n";

            cout << "4. Year Recorded: " << city.yearRecorded << "\n";

            cout << "5. Mayor: " << city.mayor << "\n";

            cout << "6. Mayor Address: " << city.mayorAddress << "\n";

            cout << "7. Short History: " << city.shortHistory << "\n";

            cout << "8. Latitude: " << city.coords.latitude << "\n";

            cout << "9. Longitude: " << city.coords.longitude << "\n";

            cout << "\nEnter the number of the field you want to update (or 0 to cancel): ";

            int field;

            cin >> field;

            cin.ignore(); // Clear newline from input buffer

            if (field == 0)

            {

                cout << "Update cancelled.\n";

                return;

            }

            switch (field)

            {

            case 1:

            {

                cout << "Enter new name: ";

                string newName;

                getline(cin, newName);

                city.name = trim(newName);

                break;

            }

            case 2:

            {

                cout << "Enter new state/country: ";

                string newState;

                getline(cin, newState);

                city.state\_or\_country = trim(newState);

                break;

            }

            case 3:

            {

                cout << "Enter new population: ";

                int newPopulation;

                cin >> newPopulation;

                cin.ignore();

                city.population = newPopulation;

                break;

            }

            case 4:

            {

                cout << "Enter new year: ";

                int newYear;

                cin >> newYear;

                cin.ignore();

                city.yearRecorded = newYear;

                break;

            }

            case 5:

            {

                cout << "Enter new mayor: ";

                string newMayor;

                getline(cin, newMayor);

                city.mayor = trim(newMayor);

                break;

            }

            case 6:

            {

                cout << "Enter new mayor address: ";

                string newAddress;

                getline(cin, newAddress);

                city.mayorAddress = trim(newAddress);

                break;

            }

            case 7:

            {

                cout << "Enter new short history: ";

                string newHistory;

                getline(cin, newHistory);

                city.shortHistory = trim(newHistory);

                break;

            }

            case 8:

            {

                cout << "Enter new latitude: ";

                double newLatitude;

                cin >> newLatitude;

                cin.ignore();

                city.coords.latitude = newLatitude;

                break;

            }

            case 9:

            {

                cout << "Enter new longitude: ";

                double newLongitude;

                cin >> newLongitude;

                cin.ignore();

                city.coords.longitude = newLongitude;

                break;

            }

            default:

                cout << "Invalid field number. Update cancelled.\n";

                return;

            }

            cout << "City details updated successfully.\n";

            return;

        }

        current = current->next; // Move to the next node

    }

    cout << "City not found.\n";

}

void CityManager::sortCitiesByYear(bool ascending)

{

    // Bubble Sort Algorithm to sort cities based on the year

    if (!cities.getHead())

        return; // If the list is empty, no need to sort

    Node \*current = cities.getHead();

    while (current)

    {

        Node \*nextNode = current->next;

        while (nextNode)

        {

            bool condition = ascending ? current->data.yearRecorded > nextNode->data.yearRecorded

                                       : current->data.yearRecorded < nextNode->data.yearRecorded;

            if (condition)

            {

                // Swap data between nodes

                City temp = current->data;

                current->data = nextNode->data;

                nextNode->data = temp;

            }

            nextNode = nextNode->next;

        }

        current = current->next;

    }

    cout << "\nCities sorted by year.\n";

}

void CityManager::displayCitiesByStateOrCountry(const string &state\_or\_country) const

{

    bool found = false;

    Node \*current = cities.getHead();

    while (current)

    {

        if (current->data.state\_or\_country == state\_or\_country)

        {

            const City &city = current->data;

            cout << fixed << setprecision(2);

            cout << "------------------------------------\n";

            cout << "Name: " << city.name << ", State/Country: " << city.state\_or\_country

                 << ", Population: " << city.population << ", Year: " << city.yearRecorded

                 << ", Mayor: " << city.mayor << ", Address: " << city.mayorAddress

                 << ", Short History: " << city.shortHistory << ", Latitude: "

                 << city.coords.latitude << ", Longitude: " << city.coords.longitude << endl;

            cout << "------------------------------------\n";

            found = true;

        }

        current = current->next;

    }

    if (!found)

    {

        cout << "No cities found in " << state\_or\_country << ".\n";

    }

}

void CityManager::displayCitiesWithSameNameDifferentCountries() const

{

    bool found = false;

    Node \*current1 = cities.getHead();

    while (current1)

    {

        Node \*current2 = current1->next;

        while (current2)

        {

            // If the cities have the same name but are located in different states or countries

            if (current1->data.name == current2->data.name &&

                current1->data.state\_or\_country != current2->data.state\_or\_country)

            {

                if (!found)

                {

                    cout << "Cities with the same name in different countries/states:\n";

                    found = true;

                }

                // Display the details of both cities

                cout << "-------------------------------------------\n";

                cout << "Name: " << current1->data.name << ", State/Country: " << current1->data.state\_or\_country

                     << ", Population: " << current1->data.population << ", Year: " << current1->data.yearRecorded

                     << ", Mayor: " << current1->data.mayor << ", Address: " << current1->data.mayorAddress

                     << ", Short History: " << current1->data.shortHistory << ", Latitude: "

                     << current1->data.coords.latitude << ", Longitude: " << current1->data.coords.longitude << endl;

                cout << "Name: " << current2->data.name << ", State/Country: " << current2->data.state\_or\_country

                     << ", Population: " << current2->data.population << ", Year: " << current2->data.yearRecorded

                     << ", Mayor: " << current2->data.mayor << ", Address: " << current2->data.mayorAddress

                     << ", Short History: " << current2->data.shortHistory << ", Latitude: "

                     << current2->data.coords.latitude << ", Longitude: " << current2->data.coords.longitude << endl;

                cout << "-------------------------------------------\n";

            }

            current2 = current2->next;

        }

        current1 = current1->next;

    }

    if (!found)

    {

        cout << "No cities with the same name found in different states or countries.\n";

    }

}

**Coordinates.h:**

#ifndef COORDINATES\_H

#define COORDINATES\_H

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

// Coordinates structure to store latitude and longitude

struct Coordinates

{

    double latitude;

    double longitude;

    // Constructor with default values

    Coordinates(double lat = 0.0, double lon = 0.0);

    // Calculate the distance between two coordinates (Haversine formula)

};

#endif // COORDINATES\_H

**Coordinates.cpp:**

#include "Coordinates.h"

#include <cmath>

Coordinates::Coordinates(double lat, double lon) : latitude(lat), longitude(lon) {}

**Utility.h:**

#ifndef UTILITY\_H

#define UTILITY\_H

#include <algorithm>

#include <string>

// Helper function to trim leading and trailing spaces

inline std::string trim(const std::string &str)

{

    const auto begin = str.find\_first\_not\_of(" \t");

    if (begin == std::string::npos)

        return ""; // No non-whitespace characters

    const auto end = str.find\_last\_not\_of(" \t");

    return str.substr(begin, end - begin + 1);

}

#endif // UTILITY\_H

**Node.h:**

#ifndef NODE\_H

#define NODE\_H

#include "City.h"

class Node

{

public:

    City data;  // The city object

    Node \*next; // Pointer to the next node

    // Constructor

    Node(const City &city) : data(city), next(nullptr) {}

};

#endif // NODE\_H

**LinkedList.h:**

#ifndef LINKEDLIST\_H

#define LINKEDLIST\_H

#include "Node.h"

#include <iostream>

using namespace std;

class LinkedList

{

private:

    Node \*head; // Pointer to the head node

public:

    // Constructor

    LinkedList() : head(nullptr) {}

    void setHead(Node \*newHead) { head = newHead; }

    // Destructor to free memory

    ~LinkedList()

    {

        clear();

    }

    Node \*getHead() const { return head; }

    // Add a city to the list

    void add(const City &city)

    {

        // Check for duplicates

        if (find(city.name, city.state\_or\_country))

        {

            cout << "Duplicate city detected: " << city.name << " in " << city.state\_or\_country << ". Not added.\n";

            return;

        }

        Node \*newNode = new Node(city);

        newNode->next = head;

        head = newNode;

        cout << "City " << city.name << " (" << city.state\_or\_country << ") added successfully!\n";

    }

    void loaded(const City &city)

    {

        Node \*newNode = new Node(city);

        newNode->next = head;

        head = newNode;

    }

    // Find a city in the list

    Node \*find(const string &name, const string &state\_or\_country) const

    {

        Node \*current = head;

        while (current)

        {

            if (current->data.name == name && current->data.state\_or\_country == state\_or\_country)

            {

                return current;

            }

            current = current->next;

        }

        return nullptr;

    }

    // Clear all nodes in the list

    void clear()

    {

        while (head)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

        }

    }

};

#endif // LINKEDLIST\_H

**Main.cpp:**

#include <iostream>

#include "CityManager.h"

using namespace std;

int main()

{

    CityManager manager;

    // Load data from file at the start

    manager.loadFromFile();

    string command;

    cout << "City Manager Program\n";

    cout << "------------------------------------------------------------------\n";

    cout << "Available commands:\n";

    cout << "1. add <name>, <state\_or\_country>, <population>, <year>, <mayor>, <mayor\_address>, <short\_history>, <latitude>, <longitude>\n";

    cout << "2. delete <name>, <state\_or\_country>\n";

    cout << "3. display\n";

    cout << "4. modify <city\_name>, <state\_or\_country>\n";

    cout << "5. distance <name\_city1>, <state\_or\_country\_1>, <name\_city2>, <state\_or\_country\_2>\n";

    cout << "6. sort\_by\_year\n";

    cout << "7. search <country/state> (It will show all the cities in a country/state) \n";

    cout << "8. duplicate (it will show all cities with same names but in different countries)\n";

    cout << "9. save\n";

    cout << "10. quit\n";

    cout << "------------------------------------------------------------------\n";

    while (true)

    {

        cout << "\nEnter command: ";

        getline(cin, command);

        istringstream input(command);

        string action;

        input >> action;

        if (action == "add")

        {

            string name, state\_or\_country, mayor, mayor\_address, short\_history;

            int population = 0, year = 0;

            double latitude = 0.0, longitude = 0.0;

            try

            {

                // Parse and validate each field

                getline(input, name, ',');

                if (trim(name).empty())

                    throw invalid\_argument("City name cannot be empty.");

                getline(input, state\_or\_country, ',');

                if (trim(state\_or\_country).empty())

                    throw invalid\_argument("State or country cannot be empty.");

                if (!(input >> population) || population <= 0)

                    throw invalid\_argument("Population must be a positive integer.");

                input.ignore(); // Ignore comma

                if (!(input >> year) || year <= 0)

                    throw invalid\_argument("Year must be a positive integer.");

                input.ignore(); // Ignore comma

                getline(input, mayor, ',');

                if (trim(mayor).empty())

                    throw invalid\_argument("Mayor cannot be empty.");

                getline(input, mayor\_address, ',');

                if (trim(mayor\_address).empty())

                    throw invalid\_argument("Mayor address cannot be empty.");

                getline(input, short\_history, ',');

                if (trim(short\_history).empty())

                    throw invalid\_argument("Short history cannot be empty.");

                if (!(input >> latitude) || latitude < -90.0 || latitude > 90.0)

                    throw invalid\_argument("Latitude must be between -90 and 90 degrees.");

                input.ignore(); // Ignore comma

                if (!(input >> longitude) || longitude < -180.0 || longitude > 180.0)

                    throw invalid\_argument("Longitude must be between -180 and 180 degrees.");

                // If all fields are valid, create the City object and add it

                City city(trim(name), trim(state\_or\_country), population, year, trim(mayor), trim(mayor\_address), trim(short\_history), Coordinates(latitude, longitude));

                manager.addCity(city);

            }

            // Catch any validation errors and notify the user

            catch (const exception &e)

            {

                cout << "Error adding city: " << e.what() << "\n";

            }

        }

        else if (action == "delete")

        {

            string name, state\_or\_country;

            getline(input, name, ',');

            getline(input, state\_or\_country, ',');

            manager.deleteCity(trim(name), trim(state\_or\_country));

        }

        else if (action == "display")

        {

            manager.displayAll(); // Default: Read from memory

        }

        else if (action == "modify")

        {

            string name, state\_or\_country;

            getline(input, name, ',');

            getline(input, state\_or\_country, ',');

            manager.modifyCity(trim(name), trim(state\_or\_country));

        }

        else if (action == "distance")

        {

            string name1, state1, name2, state2;

            // Read the parameters separated by commas

            getline(input, name1, ',');

            getline(input, state1, ',');

            getline(input, name2, ',');

            getline(input, state2, ',');

            // Pass trimmed values to the calculateDistance method

            manager.calculateDistance(trim(name1), trim(state1), trim(name2), trim(state2));

        }

        else if (action == "search")

        {

            string state\_or\_country;

            getline(input, state\_or\_country);          // Get the rest of the input after "search"

            state\_or\_country = trim(state\_or\_country); // Trim spaces around the input

            // If the state or country is provided, display cities in that state or country

            if (!state\_or\_country.empty())

            {

                manager.displayCitiesByStateOrCountry(state\_or\_country); // Display cities in the state or country

            }

            else

            {

                cout << "Invalid format. Please use the format: search <state/country>\n";

            }

        }

        else if (action == "duplicate")

        {

            manager.displayCitiesWithSameNameDifferentCountries();

        }

        else if (action == "sort\_by\_year")

        {

            manager.sortCitiesByYear(true); // Sort cities by year in ascending order

        }

        else if (action == "save")

        {

            manager.saveToFile();

        }

        else if (action == "quit")

        {

            manager.saveToFile();

            cout << "Exiting the program. Goodbye!\n";

            break;

        }

        else

        {

            cout << "Unknown command. Please try again.\n";

        }

    }

    return 0;

}

**CMakeLists.txt:**

cmake\_minimum\_required(VERSION 3.10)

project(CitiesProject)

# Specify the C++ standard

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED True)

# Add the source files

set(SOURCES

main.cpp

City.cpp

CityManager.cpp

Coordinates.cpp

LinkedList.h

Node.h

Utility.h

)

# Add the executable

add\_executable(CitiesProject ${SOURCES})