CS4414 Fall 2024: Homework Problem 2

Overview

The goal of this assignment is to develop a C++ program that processes a data set containing population statistics for cities around the world (data source: https://www.kaggle.com/datasets/dataanalyst001/world-population-growth-rate-by-cities-2024). The program reads data from a CSV file and loads city data, one city at a time, into a **singly linked list** that keeps nodes sorted according to the **city's 2024 population in descending order**. You will implement various functionalities to manipulate and analyze this data, such as sorting, filtering, and exporting specific ranges of cities. This assignment will help you gain experience with C++ classes, pointers, linked list data structures, file I/O, and basic memory management.

The assignment is broken down into three main tasks:

1. Develop the CityNode class. Due on 09/15
2. Develop the SinglyLinkedList class. Due on 09/22
3. Parse data from a CSV file into the linked list. Export selected data to a CSV file using command-line arguments. Due on 09/29

For each task, we have provided you with starter code and unit tests. You need to submit the code to Gradescope with code finished at each due date.

Programming description

Part 1: Develop the `CityNode` Class
Due date: 09/15

Objective: Define a class that represents a city and stores its population data. This is the basic building block of the singly linked list, which uses shared pointers to link the city objects according to a certain order.

There is a header file `CityNode.hpp` that contains the class members' declarations, and a corresponding implementation file `CityNode.cpp`. Implement the function and declaration in these two files in the area marked with `/*** TODO: */` Your declaration of the class's fields

should be identical to the declarations below in order to pass the test cases.

Class Definition:

Define a class `CityNode` with the following public attributes:

- `std::string name`: The name of the city.
- `std::string country`: The country in which the city is located.
- `std::string continent`: The continent on which the city is located.
- `long population_2024`: The population of the city in 2024.
- `long population_2023`: The population of the city in 2023.
- `double growth_rate`: The growth rate of the population from 2023 to 2024.

The class should also include pointers to the next node in the singly linked list, set these fields as public as well:

- `std::shared_ptr<CityNode> next`: Pointer to the next city in the list.

Constructor and destructor:

- Constructor:
    - Default constructor: The default constructor is a constructor that can be called with no arguments. Implement a default constructor that doesn't take in any arguments, and assigns default values to the object's attributes (empty string for string type attributes, 0 for long type, 0.0 for double attributes, nullptr for pointer type attributes).
    - Parameterized constructor: define a parameterized constructor for this class that takes the attributes as inputs, and assigns them to the corresponding fields.
- Destructor:
    - A destructor is a special member function that is called when the lifetime of an object ends. It should free the resources that the object may have acquired during its lifetime. In this case, we do not need a custom destructor for the CityNode class. The default destructor provided by the compiler will suffice.

Testing Implementation:

- We provided a basic test case for you to unit test your implementation of this part, in part1_test.cpp.
    - You can compile the test using the command:

```
g++ -std=c++2a -Wall CityNode.cpp part1_test.cpp -o part1_test
```

    - You can run the test using:

```
./part1_test
```

- If your implementation is correct, it prints out:

```
Part1 tests passed successfully.
```

- Note that the test file provided a preliminary test example. You can add more test

cases to thoroughly test your program.

Part 2: Develop the `SinglyLinkedList` Class
Due date 09/21

Objective: Implement a singly linked list to manage the city data. Objects are ordered according to their **population_2024 field in descending order**, with the head node having the largest population number in 2024.

Background:

1. Singly linked list
   The singly linked list consists of a list of nodes. Each node is connected to its next node by a pointer: next, which points to the next node in the list. With a singly linked list data structure, it is efficient to traverse the list in one direction, and easy to insert or delete data at any location in the list.

2. Memory allocation and management:
   In this project, we utilize **heap memory** to store the node objects. Unlike stack-allocated memory, which would automatically release the memory when the object goes out of scope, memory allocated on a heap needs to be explicitly freed when the object life cycle ends.

   We normally make sure this will occur by using a std::shared_ptr<type> object as a wrapper (container) for our malloc-ed objects. This kind of object will automatically track how many references exist to a pointer, and free the memory associated with it when the count drops to 0.

   (*** Optional: The best designs for your class would follow the RAII principle, which you can learn more about here: RAII principle https://en.cppreference.com/w/cpp/language/raii ***)

Class Definition:

In the starter code, we provided you with a header file `SinglyLinkedList.hpp` and a corresponding implementation file `SinglyLinkedList.cpp`.

Define a class `SinglyLinkedList` with the following attributes:

- `std::shared_ptr<CityNode> head`: Shared pointer to the first node in the list.

Constructor and destructor:

- Default constructor: The default constructor assigns nullptr to the head and tail attributes.

- Destructor: Since we use smart pointers to handle the objects on heap, the default destructor provided by the compiler will suffice.

Core Methods:

- `void insertCity(std::string name, std::string country, std::string continent, long population_2024, long population_2023, double growth_rate)`: Inserts a city into the list, maintaining the list in **descending order of population in 2024**.
    - Creates the `CityNode` object on the heap with **`std::make_shared`** .
    - Starting from the head of the singly linked list, traverse the list until finding the appropriate position in the list.
    - Reassign the next pointers of the nodes before this newly inserted node. Assign its next pointer to its next node if it isn't at the end of the list.
    - If two cities have the same population in 2024, then sort them by descending order of 2023 population. If they have the same population for both 2024 and 2023, sort the cities by their names in descending alphabetical order based on their first letters. (You may assume that no two cities have the same populations of 2024, 2023 and the same first letters)
- `void deleteCity(std::string name)`: Deletes a CityNode from the list with the name in the argument. Similar to insertCity, implement the pointer reassignment to keep the rest of the list connected. If no CityNode in the list has the same name as the argument, then this function should do nothing.
- `void printList() const`: Prints all the CityNode objects in the list. We will not grade you on this function. The main purpose for this function is for you to debug your code.

Range Access Methods:

- Implement methods to access specific parts of the list:
    - `std::shared_ptr<CityNode> getFirstCity() const`: Returns the first city in the list. Return nullptr if the list is empty.
    - `std::shared_ptr<CityNode> getLastCity() const`: Returns the last city in the list. Return nullptr if the list is empty.
    - `std::shared_ptr<CityNode> getCityAtIndex(int index) const`: Returns the city at a specific index. **Indices start from 1.** The city with the highest 2024 population has index 1, second highest population with index 2, etc. For example, function call `getCityAtIndex(10)` returns the shared pointer to CityNode of the 10th highest 2024 population. Return nullptr if the list is empty or if the index exceeds the total number of cities of the singly linked list.

Testing Implementation:

Similar to Part1, we provide you the test program for this part. You can compile your program by

```
g++ -std=c++2a -Wall -o part2_test part2_test.cpp CityNode.cpp
```

`SinglyLinkedList.cpp`

Note that the test file provided a preliminary test example. You can add more test cases to thoroughly test your program.

Memory leak check:

- In lecture and recitation, we learnt about a tool to find memory leaks, **Valgrind**. In this assignment, you can use this tool to ensure your program doesn't contain any memory leaks.
- Valgrind is a tool for dynamic analysis of the memory management for your program. If you are using ugclinux server to program your assignment, it has been installed and configured there, such that you can directly use it. (You can refer to recitation1, with more detailed instructions on how to program on ugclinux server)
- To run valgrind, you can run your program with `valgrind --leak-check=full` . For example, for Part 2, you can run the test program as below to check for memory usage

```
valgrind --leak-check=full ./part2_test
```

- If you see output like this, then your code has no memory leaks:

```
All heap blocks were freed -- no leaks are possible.
```

Part 3: Input and output to CSV Due date: 09/29

Part 3.1: Parse Data from CSV

Objective: Load city population data from a CSV file into the singly linked list.

The CSV is formatted as below:

```
City,Country,Continent,Population(2024),Population(2023),Growth Rate
Tokyo,Japan,Asia,37115035,37194105,-0.0021
Delhi,India,Asia,33807403,32941309,0.0263
Shanghai,China,Asia,29867918,29210808,0.0225
...
```

Reading a CSV file:

- In `SinglyLinkedList.cpp`, implement the method `void loadFromCSV(const`

`std::string& filename)`:

- Open the CSV file `filename` using `std::ifstream`.
- Extract the relevant data (city name, country, continent, population, growth rate) and insert it into the linked list using the `insertCity` method.

CSV Export Method:

- In `SinglyLinkedList.cpp`, implement the method `void writeRangeOfCitiesToCsv(const std::string& filename, int start_index, int end_index) const`:
  - Create a new CSV file at `filename` using `std::ofstream`.
  - The format of the output csv should have the same columns as the input csv format, as shown. The first line denotes the names of the columns and later lines are the rows in the csv.
  - Write the cities in the specified range (from `start_index` to `end_index`) to the file. The selected cities **include** the start_index and **exclude** the end_index. For example, if start_index=1, end_index=3, the selected cities include the 1st highest 2024 population city, and the 2nd highest 2024 population city. If start_index exceeds the total number of cities, the CSV will contain only the header row. If end_index exceeds the total number of cities, the city output will include cities up to the last city in the singly linked list.

Compilation

- You can compile your program by

```
g++ -std=c++20 -Wall SinglyLinkedList.cpp CityNode.cpp main.cpp -o CityPopulationAnalysis
```

Testing:

- Test the `loadFromCSV` method using a sample CSV file to ensure that all cities are correctly loaded into the linked list. You can use the provided csv file: world_population_by_city_2024.csv

Part 3.2: Command-Line Arguments

Objective: Allow users to export a range of cities to a new CSV file based on command-line input.

Main program:

The **main.cpp** program should perform the following:

1. parse the argument to get input csv directory, output csv directory, and the range number (start position, end position) via flag. (e.g. `./CityPopulationAnalysis -i world_pupulation_by_city_2024.csv -o output.csv -s 10 -e 20`)
2. construct Singly linked list by loading it from CSV
3. write the output to the csv

Command-Line Argument Parsing:

1. Modify `main.cpp` to accept command-line arguments for specifying the input CSV file and the range of cities to export :
    - `-i <input_csv>`: Specify the input CSV file.
    - `-o <output_csv>`: Specify the csv file pathname to write to.
    - `-s <start_index>`: Specify the start index of the range of cities to export.
    - `-e <end_index>`: Specify the end index of the range of cities to export.
2. Example command to run the program
   `./CityPopulationAnalysis -i world_population_by_city_2024.csv -o output.csv -s 10 -e 20`
3. The order of these flags may be different, e.g.

   `./CityPopulationAnalysis -o output.csv -s 10 -e 20 -i world_population_by_city_2024.csv`

   `./CityPopulationAnalysis -o output.csv -e 20-i world_population_by_city_2024.csv -s 10`

   are all valid commands. So your code need to read the argument by the flag and the argument after the flag.

Example output:

For the command
`./CityPopulationAnalysis -o output.csv -s 10 -e 12 -i world_population_by_city_2024.csv`

The output.csv looks as follow:

```
City,Country,Continent,Population(2024),Population(2023),Growth Rate
Osaka,Japan,Asia,18967459,19013434,-0.0024
Chongqing,China,Asia,17773923,17340704,0.025
```

Submission Requirements and Grading

What to Submit:

- A zip file containing:
  - All source code files (`.hpp` and `.cpp` files).
  - An example output file generated by your program (e.g., `output_10_to_20_cities.csv`).

How to Submit:

- Upload your zip file to Gradescope by the due date.

Grading Criteria:

- **Correctness:** the program compiles and correctly loads the data, handles command-line arguments, and outputs the correct range of cities. You are encouraged to add more test cases to the given test files.
- **Memory safety**: program must produce no memory leaks for full credit
- For the **autograder** to run correctly, make sure you replace all **TODO** remarks with function implementations. Do not change any function declarations, input arguments, or output types. You can add additional helper functions, as long as you implement the function declared in the starter code.