```cpp
// Candlestick.cpp
// Code written by [suhun]
#include "Candlestick.h"

Candlestick::Candlestick(const std::string& date, double open, double high,
double low, double close)
    : date(date), open(open), high(high), low(low), close(close) {}

std::string Candlestick::getDate() const { return date; }
double Candlestick::getOpen() const { return open; }
double Candlestick::getHigh() const { return high; }
double Candlestick::getLow() const { return low; }
double Candlestick::getClose() const { return close; }
```

```cpp
// Candlestick.h
// Code written by [suhun]

#pragma once

#include <string>
#include <vector>

class Candlestick
{
public:
    Candlestick(const std::string& date, double open, double high, double low,
double close);

    std::string getDate() const;
    double getOpen() const;
    double getHigh() const;
    double getLow() const;
    double getClose() const;

private:
    std::string date; // Year or specific time frame
    double open;      // Open value
    double high;      // High value
    double low;       // Low value
    double close;     // Close value
};
```

```cpp
//CandlestickCalculator.cpp
// Code written by [suhun]

#include "CandlestickCalculator.h"
#include <algorithm>
#include <numeric>



std::vector<Candlestick> CandlestickCalculator::computeCandlesticks(const
std::vector<TemperatureEntry>& entries, const std::string& country)
{
    std::vector<Candlestick> candlesticks;
    auto groupedData = groupByYear(entries, country);

    std::string previousYear = "";
    double previousClose = 0.0;

    for (const auto& [year, values] : groupedData)
    {
        if (values.empty())
            continue;

        double open = previousClose;
        double high = *std::max_element(values.begin(), values.end());
        double low = *std::min_element(values.begin(), values.end());
        double close = std::accumulate(values.begin(), values.end(), 0.0) /
values.size();

        candlesticks.emplace_back(year, open, high, low, close);
        previousClose = close;
    }

    return candlesticks;
}

std::map<std::string, std::vector<double>>
CandlestickCalculator::groupByYear(const std::vector<TemperatureEntry>&
entries, const std::string& country)
{
```

```
    std::map<std::string, std::vector<double>> groupedData;

    for (const auto& entry : entries)
    {
        if (entry.getCountry() != country)
            continue;

        std::string year = entry.getTimestamp().substr(0, 4); // Extract year
from timestamp
        groupedData[year].push_back(entry.getValue());
    }

    return groupedData;
}
```

```
//CandlestickCalculator.h
// Code written by [suhun]

#pragma once

#include "TemperatureEntry.h"
#include "Candlestick.h"
#include <vector>
#include <string>
#include <map>

class CandlestickCalculator
{
public:
    // Computes candlestick data for a specific country and time frame
    static std::vector<Candlestick> computeCandlesticks(const
std::vector<TemperatureEntry>& entries, const std::string& country);

private:
    // Helper to group data by year
    static std::map<std::string, std::vector<double>> groupByYear(const
std::vector<TemperatureEntry>& entries, const std::string& country);
};
```

```
//CandlestickFilter.cpp
// Code written by [suhun]

#include "CandlestickFilter.h"
```

```cpp
std::vector<Candlestick> CandlestickFilter::filterByDateRange(const
std::vector<Candlestick>& candlesticks, const std::string& startDate, const
std::string& endDate)
{
    std::vector<Candlestick> filtered;
    for (const auto& candlestick : candlesticks)
    {
        if (candlestick.getDate() >= startDate && candlestick.getDate() <=
endDate)
        {
            filtered.push_back(candlestick);
        }
    }
    return filtered;
}

std::vector<Candlestick> CandlestickFilter::filterByTemperatureRange(const
std::vector<Candlestick>& candlesticks, double minTemp, double maxTemp)
{
    std::vector<Candlestick> filtered;
    for (const auto& candlestick : candlesticks)
    {
        if (candlestick.getLow() >= minTemp && candlestick.getHigh() <=
maxTemp)
        {
            filtered.push_back(candlestick);
        }
    }
    return filtered;
}
```

```cpp
// CandlestickFilter.h
// Code written by [suhun]

#pragma once
#include "Candlestick.h"
#include <vector>
#include <string>

class CandlestickFilter
{
public:
    static std::vector<Candlestick> filterByDateRange(const
std::vector<Candlestick>& candlesticks, const std::string& startDate, const
std::string& endDate);
    static std::vector<Candlestick> filterByTemperatureRange(const
std::vector<Candlestick>& candlesticks, double minTemp, double maxTemp);
```

```
};
```

```
// CandlestickPlotter.cpp
// Code written by [suhun]

#include "CandlestickPlotter.h"
#include <iostream>
#include <iomanip>

// Task 2: plotting table
void CandlestickPlotter::plot(const std::vector<Candlestick>& candlesticks)
{
    const double fixedLow = -10.0; // Fixed lower bound
    const double fixedHigh = 30.0; // Fixed upper bound
    const double scaleFactor = 1.5; // Scale factor for narrower plot
    const int plotWidth = static_cast<int>((fixedHigh - fixedLow) *
scaleFactor); // Total plot width
    const int zeroScaled = static_cast<int>((0 - fixedLow) * scaleFactor); //
Zero marker position

    std::cout << "\nCandlestick Plot:\n";
    std::cout << "Date       | Plot\n";
    std::cout << "-------------------------------------------\n";

    for (const auto& candlestick : candlesticks)
    {
        double open = candlestick.getOpen();
        double high = candlestick.getHigh();
        double low = candlestick.getLow();
        double close = candlestick.getClose();
        std::string date = candlestick.getDate();

        // Adjust scale based on fixed bounds
        int highScaled = static_cast<int>((high - fixedLow) * scaleFactor);
        int openScaled = static_cast<int>((open - fixedLow) * scaleFactor);
        int closeScaled = static_cast<int>((close - fixedLow) * scaleFactor);
        int lowScaled = static_cast<int>((low - fixedLow) * scaleFactor);


        // Print date
        std::cout << std::setw(10) << date << " | ";

        // Plot the candlestick with zero position aligned
        for (int i = 0; i <= plotWidth; ++i)
        {
            if (i == zeroScaled)
```

```
                std::cout << "o"; // Zero marker
            else if (i == highScaled)
                std::cout << "-"; // High marker
            else if (i == lowScaled)
                std::cout << "-"; // Low marker
            else if (i == openScaled)
                std::cout << "|"; // Open marker
            else if (i == closeScaled)
                std::cout << "|"; // Close marker
            else
                std::cout << " ";
        }

        std::cout << "\n";
    }
}
```

```
// CandlestickPlotter.h
// Code written by [suhun]

#pragma once
#include "Candlestick.h"
#include <vector>

class CandlestickPlotter
{
public:
    static void plot(const std::vector<Candlestick>& candlesticks);
};
```

```
// CandlestickPredictor.cpp
// Code written by [suhun]

#include "CandlestickPredictor.h"
#include <numeric>

std::vector<Candlestick> CandlestickPredictor::predictMovingAverage(const
std::vector<Candlestick>& candlesticks, int windowSize)
{
    std::vector<Candlestick> predictions;

    for (size_t i = 0; i < candlesticks.size() - windowSize + 1; ++i)
    {
```

```
        double sumOpen = 0.0, sumHigh = 0.0, sumLow = 0.0, sumClose = 0.0;
        std::string date = candlesticks[i + windowSize - 1].getDate(); // Use
the last date in the window

        for (size_t j = 0; j < windowSize; ++j)
        {
            sumOpen += candlesticks[i + j].getOpen();
            sumHigh += candlesticks[i + j].getHigh();
            sumLow += candlesticks[i + j].getLow();
            sumClose += candlesticks[i + j].getClose();
        }

        double avgOpen = sumOpen / windowSize;
        double avgHigh = sumHigh / windowSize;
        double avgLow = sumLow / windowSize;
        double avgClose = sumClose / windowSize;

        // Corrected order of arguments
        predictions.emplace_back(date, avgOpen, avgHigh, avgLow, avgClose);
    }

    return predictions;
}


// CandlestickPredictor.h
// Code written by [suhun]

#pragma once
#include "Candlestick.h"
#include <vector>
#include <string>

class CandlestickPredictor
{
public:
    static std::vector<Candlestick> predictMovingAverage(const
std::vector<Candlestick>& candlesticks, int windowSize);
};
```

```
// CSVReader.cpp
// Code written by [suhun]

#include "CSVReader.h"
#include "TemperatureEntry.h"
#include <fstream>
```

```cpp
#include <sstream>
#include <iostream>

CSVReader::CSVReader() {}

std::vector<TemperatureEntry> CSVReader::readCSV(const std::string& filename)
{
    std::vector<TemperatureEntry> entries;
    std::ifstream file(filename);
    std::string line;

    if (file.is_open())
    {
        // Skip the header line
        std::getline(file, line);

        while (std::getline(file, line))
        {
            try
            {
                std::vector<std::string> tokens = tokenize(line, ',');
                TemperatureEntry entry = stringsToTemperatureEntry(tokens);
                entries.push_back(entry);
            }
            catch (const std::exception &e)
            {
                std::cerr << "CSVReader::readCSV: Invalid data line,
skipping." << std::endl;
            }
        }
        file.close();
    }
    else
    {
        std::cerr << "CSVReader::readCSV: Unable to open file " << filename <<
std::endl;
    }

    return entries;
}

std::vector<std::string> CSVReader::tokenize(const std::string& line, char
separator)
{
    std::vector<std::string> tokens;
    std::istringstream stream(line);
    std::string token;
```

```
    while (std::getline(stream, token, separator))
    {
        tokens.push_back(token);
    }

    return tokens;
}

TemperatureEntry CSVReader::stringsToTemperatureEntry(const
std::vector<std::string>& tokens)
{
    if (tokens.size() < 3)
    {
        throw std::runtime_error("Insufficient tokens to create
TemperatureEntry");
    }

    try
    {
        std::string timestamp = tokens[0];
        std::string country = "GB";
        double value = std::stod(tokens[12]);

        return TemperatureEntry(value, timestamp, country,
EntryType::Temperature);
    }
    catch (const std::exception &e)
    {
        throw std::runtime_error("Error parsing TemperatureEntry: " +
std::string(e.what()));
    }
}
```

```
// CSVReader.h
// Code written by [suhun]

#pragma once
#include "TemperatureEntry.h"
#include <vector>
#include <string>

class CSVReader
{
public:
    CSVReader();

    // Reads a CSV file and returns a vector of TemperatureEntry objects
```

```
    static std::vector<TemperatureEntry> readCSV(const std::string& filename);

    // Splits a CSV line into tokens based on the separator
    static std::vector<std::string> tokenize(const std::string& line, char
separator);

private:
    // Converts a vector of strings to a TemperatureEntry
    static TemperatureEntry stringsToTemperatureEntry(const
std::vector<std::string>& tokens);
};
```

```
// TemperatureEntry.cpp
// Code written by [suhun]

#include "TemperatureEntry.h"

TemperatureEntry::TemperatureEntry(double value, const std::string& timestamp,
const std::string& country, EntryType type)
    : value(value), timestamp(timestamp), country(country), type(type) {}

double TemperatureEntry::getValue() const { return value; }
std::string TemperatureEntry::getTimestamp() const { return timestamp; }
std::string TemperatureEntry::getCountry() const { return country; }
EntryType TemperatureEntry::getType() const { return type; }
```

```
// TemperatureEntry.h
// Code written by [suhun]

#pragma once

#include <string>
#include <vector>

enum class EntryType { Temperature };

class TemperatureEntry
{
public:
    TemperatureEntry(double value, const std::string& timestamp, const
std::string& country, EntryType type);

    double getValue() const;
```

```
    std::string getTimestamp() const;
    std::string getCountry() const;
    EntryType getType() const;

private:
    double value;          // Temperature value
    std::string timestamp; // Time of the reading
    std::string country;   // Country code
    EntryType type;        // Type of the entry
};
```

```
//main.cpp
// Code written by [suhun]

#include "CSVReader.h"
#include "CandlestickCalculator.h"
#include "TemperatureEntry.h"
#include "Candlestick.h"
#include "CandlestickPlotter.h"
#include "CandlestickFilter.h"
#include "CandlestickPredictor.h"
#include <iostream>
#include <vector>
#include <string>

int main()
{
    // Load CSV data
    std::string filename = "weather_data_EU_1980-2019_temp_only.csv";
    std::vector<TemperatureEntry> entries = CSVReader::readCSV(filename);

    // Task 1: Compute candlestick data for a specific country
    std::string country = "GB";
    std::vector<Candlestick> candlesticks =
CandlestickCalculator::computeCandlesticks(entries, country);

    // Task 3: Filter by date range
    std::string startDate = "1985-01-01";
    std::string endDate = "2019-12-31";
    std::vector<Candlestick> filteredByDate =
CandlestickFilter::filterByDateRange(candlesticks, startDate, endDate);

    // Task 3: Filter by temperature range
    double minTemp = -5.0;
    double maxTemp = 25.0;
    std::vector<Candlestick> filteredByTemp =
CandlestickFilter::filterByTemperatureRange(filteredByDate, minTemp, maxTemp);
```

```
    // Print filtered candlestick data (Task 3 result)
    std::cout << "\nFiltered Candlestick Data (by date range and
temperature):\n";
    // Print data (Task 1 result)
    for (const auto& candlestick : filteredByTemp)
    {
        std::cout << "Date: " << candlestick.getDate()
                  << ", Open: " << candlestick.getOpen()
                  << ", High: " << candlestick.getHigh()
                  << ", Low: " << candlestick.getLow()
                  << ", Close: " << candlestick.getClose() << std::endl;
    }

    // Plot the filtered candlestick data (Task 3)
    std::cout << "\nFiltered Candlestick Plot:\n";
    CandlestickPlotter::plot(filteredByTemp);

    // Task 4: Generate predicted data using moving average
    int windowSize = 3; // Moving average window size
    std::vector<Candlestick> predictions =
CandlestickPredictor::predictMovingAverage(filteredByTemp, windowSize);

    // Print predicted candlestick data (Task 4 result)
    std::cout << "\nPredicted Candlestick Data (using Moving Average):\n";
    for (const auto& candlestick : predictions)
    {
        std::cout << "Date: " << candlestick.getDate()
                  << ", Open: " << candlestick.getOpen()
                  << ", High: " << candlestick.getHigh()
                  << ", Low: " << candlestick.getLow()
                  << ", Close: " << candlestick.getClose() << std::endl;
    }

    // Plot the predicted candlestick data (Task 4)
    std::cout << "\nPredicted Candlestick Plot:\n";
    // Plotting data (Task 2)
    CandlestickPlotter::plot(predictions);

    return 0;
}
```