Code

*** Code in black I personally wrote and code in red I used assistance from the merkelrex***

CsvReader.h

```
// Ensure this header file is included only once
#pragma once

// standard library includes
#include <vector>
#include <fstream>
#include <filesystem>
#include <string>

// project header
#include "TemperatureRow.h"


class CsvReader
{
public:
	/** Takes a string csv/txt file name as input and returns
	a vector of each row as type TemperatureRow */
	static const std::vector<TemperatureRow> readcsv(const std::string fileName);
```

```
private:
        /** Takes a string csv/txt file name as input and returns
        a path to the file */
        static const std::filesystem::path getFilePath(const std::string fileName);
        // the tokenize function used in the Merkel currency exchange app
        static std::vector<std::string> tokenize(std::string csvLine, char delimiter);
        /** Takes a vector containing row data points and returns a type TemperatureRow
        containing a utcTimestamp and a vector of temperatures of type double.
        Excludes the csv file header */
        static TemperatureRow stringsToTempRow(std::vector<std::string> rowTokens);
};
```

CsvReader.cpp

```
#include "CsvReader.h"

// standard library includes
#include <vector>
#include <fstream>
#include <filesystem>
#include <string>

// project header
#include "TemperatureRow.h"

// Opens and reads the csv file, calls the tokenize() and stringsToTempRow()
// methods and returns a vector of class type TemperatureRow
const std::vector<TemperatureRow> CsvReader::readcsv(const std::string fileName)
{

        std::vector<TemperatureRow> rows;
        // create a path to the csv file using the getFilePath method
        const std::filesystem::path filePath = getFilePath(fileName);
        // pass std::filesystem::path to the ifstream constructor - uses implicit conversion to std::string
        std::ifstream csvFile{ filePath };
        std::string line;

        if (csvFile.is_open()) {
                // ignore the header line
                std::getline(csvFile, line);
```

```
                // get the rest of the rows
                // add each row as type TemperatureRow to the "rows" vector
                while (std::getline(csvFile, line)) {
                        try {
                                TemperatureRow row = stringsToTempRow(tokenize(line, ','));
                                rows.push_back(row);
                        }
                        catch (const std::exception e) {
                                std::cout << "Bad line" << std::endl;
                        }
                }
        }
        // Print the amount of rows read in.
        std::cout << "The dataset was successfully read in with " << rows.size() << " rows." << std::endl;
        std::cout << "Check the About section for dataset information." << std::endl;

        return rows;
}


// Wrote this function myself, it builds a path to the csv file from the root path
// Returns the file path if the file exists
const std::filesystem::path CsvReader::getFilePath(const std::string fileName)
{
        std::filesystem::path rootPath = std::filesystem::current_path();
        std::filesystem::path filePath = rootPath / "data" / fileName;

        if (!std::filesystem::exists(filePath))
        {
                throw std::runtime_error("Path does not exist: " + filePath.string());
        }

        return filePath;
}


// Added a line to remove any whitespace including \n \t \r in each token
std::vector<std::string> CsvReader::tokenize(std::string csvLine, char delimiter)
{
        std::vector<std::string> tokens;
        std::string token;
```

```
	signed int start, end;
	start = csvLine.find_first_not_of(delimiter, 0);

	do {
		end = csvLine.find_first_of(delimiter, start);
		if (start == csvLine.length() || start == end) break;
		if (end >= 0) token = csvLine.substr(start, end - start);
		else token = csvLine.substr(start, csvLine.length() - start);
		// Added a removal of any whitewpace
		token.erase(std::remove_if(token.begin(), token.end(), ::isspace), token.end());
		tokens.push_back(token);
		start = end + 1;
	} while (end != std::string::npos);

	return tokens;
}



TemperatureRow CsvReader::stringsToTempRow(std::vector<std::string> rowTokens)
{
	std::string utcTimestamp;
	std::vector<double> temps;

	// The row must have 29 tokens
	if (rowTokens.size() != 29) {
		std::cerr << "Error: Line has " << rowTokens.size() << " tokens\n";
		throw std::runtime_error("Invalid number of tokens");
	}

	// Move the first token (timestamp) from the rowTokens to avoid copying
	utcTimestamp = std::move(rowTokens[0]);

	// Extract the remaining tokens (temperature values) into stringTemps
	std::vector<std::string> stringTemps(rowTokens.begin() + 1, rowTokens.end());
	for (const auto& t : stringTemps) {
		try
		{
			temps.push_back(std::stod(t));
		}
		catch (const std::exception& e)
		{
```

```
                    std::cout << "Bad double" << std::endl;
                    throw e;
                }
            }
            // Create a TemperatureRow object, move both the timestamp and the temperatures to avoid
copying
            TemperatureRow temperatureRow {
                std::move(utcTimestamp),
                std::move(temps)
            };

            return temperatureRow;
}
```

TemperatureRow.h

```
// Ensure this header file is included only once
#pragma once

// standard library includes
#include <string>
#include <map>
#include <vector>
#include <iostream>


class TemperatureRow
{
// Declare the constructor with `const` references to avoid copying and ensure the arguments are not
modified.
public:
   TemperatureRow(std::string _timestamp, std::vector<double> _temperatures);

   // Getter for the TemperatureRow temperatures
   // Returns a const reference to the temperatures vector to avoid copying while allowing read-only
access.
   const std::vector<double>& getTemperatures() const;

   // Finds and returns the index of the row matching the specified timestamp from the vector of rows.
   // Static because it operates on a collection of TemperatureRow objects, not a specific instance.
```

```
    static const int getRowIndex(const std::vector<TemperatureRow>& rows, const std::string&
timestamp);
    // Groups temperatures by year and returns a map where keys are years (as strings)
    // and values are vectors of temperatures of type double. The `countryIndex` determines which
country's data to group.
    static std::map <std::string, std::vector<double>> getTempsByYear(const
std::vector<TemperatureRow>& rows,
        unsigned int countryIndex);
    // Groups temperatures by a day of the year and returns a map where keys are years (as strings)
    // and values are vectors of temperatures of type double. The `countryIndex` determines which
country's data to group.
    static std::map <std::string, std::vector<double>> getTempsByDayOfYear(const
std::vector<TemperatureRow>& rows,
        unsigned int countryIndex, const std::string monthDay);

    // Static map to associate year strings with integer values, providing a lookup for available years.
    static const std::map<std::string, unsigned int> years;
    // Static map to associate country names (as strings) with unsigned integer indices,
    // providing a lookup for country-specific data.
    static const std::map<std::string, unsigned int> countries;

private:
    std::string timestamp;
    std::vector<double> temperatures;

    std::string getYear() const;
    std::string getMonth() const;
    std::string getDay() const;
};
```

TemperatureRow.cpp

```
// Related header
#include "TemperatureRow.h"

// standard library includes
#include <cmath>
```

```
TemperatureRow::TemperatureRow(std::string _timestamp, std::vector<double> _temperatures)
   : timestamp(std::move(_timestamp)),   // Move timestamp into the member
            temperatures(std::move(_temperatures)) // Move temperatures into the member
{

}


const std::vector<double>& TemperatureRow::getTemperatures() const
{
        return TemperatureRow::temperatures;
}

// Uses binary search to locate a timestamp in a vector of temperature rows sorted by timestamp.
const int TemperatureRow::getRowIndex(const std::vector<TemperatureRow>& rows,
        const std::string& searchTimestamp)
{
        // Initialize pointers to the left most and right most indices
        int left = 0, right = int(rows.size() - 1);

        // Continue searching while the left pointer is still less than or equal to the right pointer.
        while (left <= right) {

                // Get the middle index and element.
                int mid = int(floor((left + right) / 2));
                std::string tsMid = rows[mid].timestamp;

                // Return row index if found.
                if (searchTimestamp == tsMid)
                        return mid;
                // If the search timestamp is less than the midpoint timestamp, adjust the search range
                // to exclude the right half.
                if (searchTimestamp < tsMid)
                        right = mid - 1;
                // If the search timestamp is greater than the midpoint timestamp, adjust the search
range
                // to exclude the left half.
                else
                        left = mid + 1;
        }
```

```
	// Return -1 if not found.
	return -1;
}


// Groups temperature data by year from a vector of TemperatureRow objects.
// For each year, it creates a mapping of the year (as a string) to a vector of temperatures
// for the specified country index.
std::map <std::string, std::vector<double>> TemperatureRow::getTempsByYear(const
std::vector<TemperatureRow>& rows,
	unsigned int countryIndex)
{
	// Initialize the map to store year-to-temperatures mapping.
	std::map<std::string, std::vector<double>> yearTemps;
	// Start with the year of the first row and an empty vector of temperatures.
	std::string currentYear = rows[0].getYear();
	std::vector<double> temps;

	// Iterate over each TemperatureRow in the rows vector.
	for (const auto& row : rows) {
		// Check the current year still matches the row year and add the row temperature for
the chosen country.
		if (currentYear == row.getYear()) {
			temps.push_back(row.temperatures[countryIndex]);
		}
		// The row year has moved to the next year and does not match the current year.
		// Map the current year to a vector of temperatures from that year for the chosen
country.
		// Update the current year, empty the vector and add the current row as the first
temperature
		// for the next year.
		else {
			yearTemps[currentYear] = temps;
			currentYear = row.getYear();
			temps.clear();
			temps.push_back(row.temperatures[countryIndex]);
		}
	}
	// Add the final years temps to the map (2019)
	yearTemps[currentYear] = temps;

	return yearTemps;
```

```
}


// Groups temperature data by a day of the year from a vector of TemperatureRow objects.
// For each year, it creates a mapping of the year (as a string) to a vector of temperatures
// for the specified country index.
std::map <std::string, std::vector<double>> TemperatureRow::getTempsByDayOfYear(const
std::vector<TemperatureRow>& rows,
	unsigned int countryIndex, const std::string monthDay)
{
	// Initialize the map to store year-to-temperatures mapping.
	std::map<std::string, std::vector<double>> dayOfYearTemps;
	// Start with the year of the first row and an empty vector of temperatures.
	std::string currentYear = rows[0].getYear();
	std::vector<double> temps;

	// Iterate over each TemperatureRow in the rows vector.
	for (const auto& row : rows) {
		// Check the current year still matches the row year.
		if (currentYear == row.getYear()) {
			// Check the month and day matches the chosen day and add the row
temperature for the chosen country.
			if (row.getMonth() + "-" + row.getDay() == monthDay) {
				temps.push_back(row.temperatures[countryIndex]);
			}
		}
		// The row year has moved to the next year and does not match the current year.
		// Map the current year to a vector of temperatures from that day of the year for the
chosen country.
		// Update the current year and empty the vector.
		else {
			dayOfYearTemps[currentYear] = temps;
			currentYear = row.getYear();
			temps.clear();
			// If the first row of the next year matches the month and day (this woulod be
January 1st at 00:00),
			// add this row to the empty vector
			if (row.getMonth() + "-" + row.getDay() == monthDay) {
				temps.push_back(row.temperatures[countryIndex]);
			}
		}
	}
```

```
        // Add the final years temps to the map (2019)
        dayOfYearTemps[currentYear] = temps;

        return dayOfYearTemps;
}


const std::map<std::string, unsigned int> TemperatureRow::years = {
        {"1980", 0},
        {"1981", 1},
        {"1982", 2},
        {"1983", 3},
        {"1984", 4},
        {"1985", 5},
        {"1986", 6},
        {"1987", 7},
        {"1988", 8},
        {"1989", 9},
        {"1990", 10},
        {"1991", 11},
        {"1992", 12},
        {"1993", 13},
        {"1994", 14},
        {"1995", 15},
        {"1996", 16},
        {"1997", 17},
        {"1998", 18},
        {"1999", 19},
        {"2000", 20},
        {"2001", 21},
        {"2002", 22},
        {"2003", 23},
        {"2004", 24},
        {"2005", 25},
        {"2006", 26},
        {"2007", 27},
        {"2008", 28},
        {"2009", 29},
        {"2010", 30},
        {"2011", 31},
        {"2012", 32},
```

```
            {"2013", 33},
            {"2014", 34},
            {"2015", 35},
            {"2016", 36},
            {"2017", 37},
            {"2018", 38},
            {"2019", 39}
};


const std::map<std::string, unsigned int> TemperatureRow::countries = {
   {"AUSTRIA", 0},
   {"BELGIUM", 1},
   {"BULGARIA", 2},
   {"SWITZERLAND", 3},
   {"CZECH REPUBLIC", 4},
   {"GERMANY", 5},
   {"DENMARK", 6},
   {"ESTONIA", 7},
   {"SPAIN", 8},
   {"FINLAND", 9},
   {"FRANCE", 10},
   {"UNITED KINGDOM", 11},
   {"GREECE", 12},
   {"CROATIA", 13},
   {"HUNGARY", 14},
   {"IRELAND", 15},
   {"ITALY", 16},
   {"LITHUANIA", 17},
   {"LUXEMBOURG", 18},
   {"LATVIA", 19},
   {"NETHERLANDS", 20},
   {"NORWAY", 21},
   {"POLAND", 22},
   {"PORTUGAL", 23},
   {"ROMANIA", 24},
   {"SWEDEN", 25},
   {"SLOVENIA", 26},
   {"SLOVAKIA", 27}
};
```

```
////////////////// Private methods ///////////////////


// Extracts and returns the year component from the timestamp string.
std::string TemperatureRow::getYear() const {
        // Extracts the first 4 characters representing the year.
   return timestamp.substr(0, 4);
}

// Extracts and returns the month component from the timestamp string.
std::string TemperatureRow::getMonth() const {
        // Extracts the characters at positions 5 and 6, representing the month.
   return timestamp.substr(5, 2);
}


// Extracts and returns the day component from the timestamp string.
std::string TemperatureRow::getDay() const {
        // Extracts the characters at positions 8 and 9, representing the day.
   return timestamp.substr(8, 2);
}
```

Candlestick.h

```
// Ensure this header file is included only once
#pragma once

// standard library includes
#include <string>
#include <map>
#include <vector>


class Candlestick
{
public:
        Candlestick(std::string _year, double _open, double _close, double _high, double _low);
```

```
        // Getters for each member
        const std::string& getYear() const;
        double getOpen() const;
        double getClose() const;
        double getHigh() const;
        double getLow() const;

        // Static function that generates a candlestick chart.
        // It takes a vector of Candlestick objects and returns a map where:
        // - The key is the y axis
        // - The value is a string line at that height on the y axis
        static std::map<int, std::string, std::greater<int>> getCandlestickChart(
                const std::vector<Candlestick>& candlesticks);

private:
        // Helper function that calculates the Y-axis values for the candlestick chart.
        static std::map<int, std::string, std::greater<int>> calculateYAxis(const std::vector<Candlestick>&
candlesticks);
        const std::string year;
        double open;
        double close;
        double high;
        double low;
};
```

Candlestick.cpp

```
#include "Candlestick.h"
#include "Statistics.h"

#include <cmath>


Candlestick::Candlestick(std::string _year, double _open, double _close, double _high, double _low)
   : year(_year), open(_open), close(_close), high(_high), low(_low)
{
```

```
}


const std::string& Candlestick::getYear() const {
   return year;
}

double Candlestick::getOpen() const {
   return open;
}

double Candlestick::getClose() const {
   return close;
}

double Candlestick::getHigh() const {
   return high;
}

double Candlestick::getLow() const {
   return low;
}

// Returns a descending map where the key is a rounded Y-axis value and the value is a string
// line at that y-axis height for each year with a text representation of a candlstick
std::map<int, std::string, std::greater<int>> Candlestick::getCandlestickChart(
        const std::vector<Candlestick>& candlesticks)
{
        // Initialize the chart map with calculated Y-axis values.
        // The map is ordered in descending order to print from the top of the graph to the bottom
        std::map<int, std::string, std::greater<int>> chart = calculateYAxis(candlesticks);
        // Define ANSI color escape codes for terminal output.
        const std::string reset = "\033[0m";
        const std::string red = "\033[31m";
        const std::string green = "\033[32m";
        // Iterate over each candlestick in the input vector and round the open, close, high, and low.
        for (const auto& c : candlesticks) {
                int open = static_cast<int>(std::round(c.getOpen()));
                int close = static_cast<int>(std::round(c.getClose()));
                int high = static_cast<int>(std::round(c.getHigh()));
                int low = static_cast<int>(std::round(c.getLow()));
```

```
                // Iterate over each Y-axis level in the chart from top to bottom.
                for (auto& pair : chart) {
                        // If the current Y-axis level is within the open-close range, draw a filled block.
                        if (pair.first >= std::min(open, close) && pair.first <= std::max(open, close)) {
                                if (close >= open)
                                        chart[pair.first] += green + " \u2588" + " " + reset; // Green block
for positive candlestick
                                else
                                        chart[pair.first] += red + " \u2588" + " " + reset; // Red block for
positive candlestick
                        }
                        // Else if the current Y-axis level is within the low-high range, draw a vertical line.
                        else if (pair.first >= low && pair.first <= high) {
                                if (close >= open)
                                        chart[pair.first] += green + " \u2502" + " " + reset; // Green block
for positive candlestick
                                else
                                        chart[pair.first] += red + " \u2502" + " " + reset; // Red block for
positive candlestick
                        }
                        // Else add 3 empty spaces to fill the space to the next year
                        else {
                                chart[pair.first] += "   ";
                        }
                }
        }
        return chart;
}


// Returns a map with decsending y-axis values as keys and an empty string as values
std::map<int, std::string, std::greater<int>> Candlestick::calculateYAxis(const std::vector<Candlestick>&
candlesticks)
{
        // Calculate the Y-axis scale for the candlestick chart.
        // The Y-axis values are stored in a map ordered in descending order (std::greater<int>).
        std::map<int, std::string, std::greater<int>> yAxis;
        // Initialize variables to store the maximum and minimum values of the Y-axis.
        double yAxisMax = std::numeric_limits<double>::min(); // Smallest possible double value.
        double yAxisMin = std::numeric_limits<double>::max(); // Largest possible double value.
```

```
	// Iterate through the candlesticks to find the maximum and minimum Y-axis values.
	for (const auto& c : candlesticks) {
		// Find the highest and lowest values among the open, close, high, and low prices.
		double highest = std::max({ c.getOpen(), c.getClose(), c.getHigh() });
		double lowest = std::min({ c.getOpen(), c.getClose(), c.getLow() });
		// Update the maximum and minimum Y-axis values if necessary.
		if (highest > yAxisMax)
			yAxisMax = highest;
		if (lowest < yAxisMin)
			yAxisMin = lowest;
	}

	// Round the maximum and minimum Y-axis values to the nearest integers to scale the y-axis
	// and allow data points to be rounded and assigned to a key in the map
	int yAxisMaxRound = static_cast<int>(std::round(yAxisMax));
	int yAxisMinRound = static_cast<int>(std::round(yAxisMin));

	// Create the Y-axis map with integer levels from the rounded maximum to minimum values.
	// Each level is initialized with an empty string to hold chart data later.
	for (int i = yAxisMaxRound; i >= yAxisMinRound; i -= 1) {
		yAxis[i] = "";
	}

	return yAxis;
}
```

Statistics.h

```
// Ensure this header file is included only once
#pragma once

// standard library includes
#include <vector>
#include <map>
#include <string>
// project headers
#include "Candlestick.h"
```

```
#include "ScatterPlotHighLow.h"

class Statistics
{

public:
	// Calculates candlesticks (open, close, high, low) for each year from a given map of
year-to-temperatures.
	// Input: A map where keys are years (strings) and values are vectors of temperatures (doubles).
	// Output: A vector of Candlestick objects, each representing a year's temperature statistics.
	static std::vector<Candlestick> calculateCandlesticks(const std::map<std::string,
		std::vector<double>>& yearToTempsMap);
	// Calculates scatter plot data for high and low temperatures over years.
	// Input: A map where keys are years (strings) and values are vectors of temperatures (doubles).
	// Output: A vector of ScatterPlotHighLow objects, each representing high and low values for a
year.
	static std::vector<ScatterPlotHighLow> calculateScatterPlotHighLows(const std::map<std::string,
		std::vector<double>>& yearToTempsMap);
	// Calculates the correlation coefficient for a given set of data points.
	// Input: A vector of pairs, where each pair contains a year (string) and a value (double).
	// Output: A double representing the correlation coefficient, indicating the strength and
direction of the relationship.
	static double getCorrelationCoefficient(std::vector<std::pair<std::string, double>>
predictionData);
	// Predicts a temperature with simple linear regression using the least squares method.
	// Input: A vector of pairs (year as string, value as double) representing historical data and a year
(double) for prediction.
	// Output: A double representing the predicted temperature for the provided data.
	static double getLinearRegressionPrediction(std::vector<std::pair<std::string, double>>
predictionData, double year);
	// Calculates the mean, high, and low temperatures from a vector of temperatures.
	// Input: A vector of temperatures (doubles).
	// Output: A vector containing three doubles: the mean, high, and low temperatures.
	static std::vector<double> getMeanHighLow(const std::vector<double>& temps);
	// Finds the highest and lowest temperatures from a given set of temperatures.
	// Input: A vector of temperatures (doubles).
	// Output: A pair of doubles representing the highest and lowest temperatures.
	static std::pair<double, double> getHighLow(const std::vector<double>& temps);
};
```

Statistics.cpp

```
#include "Statistics.h"

// standard library includes
#include <numeric>
#include <algorithm>
#include <cmath>
#include <iostream>


std::vector<Candlestick> Statistics::calculateCandlesticks(const std::map<std::string,
	std::vector<double>>& yearToTempsMap)
{
	// Initialize an empty vector to store the resulting candlesticks.
	std::vector<Candlestick> candlesticks;

	// If the input map is empty, return an empty candlestick vector.
	if (yearToTempsMap.empty()) {
		return candlesticks;
	}

	// Handle the first year's data where the open will be the same as the close
	const auto& firstYear = *yearToTempsMap.begin(); // Get the first key-value pair in the map.
	const std::string& year = firstYear.first; // Extract the year (key) as a string.
	const std::vector<double>& firstTemps = firstYear.second; // Extract the vector of temperatures

	// Calculate the mean, high, and low temperatures for the first year.
	std::vector<double> meanHighLow = getMeanHighLow(firstTemps);
	double open = meanHighLow[0]; // current year's mean
	double close = meanHighLow[0]; // current year's mean
	double high = meanHighLow[1];
	double low = meanHighLow[2];

	// Create the first Candlestick object and add it to the vector.
	Candlestick candlestick(year, open, close, high, low);
	candlesticks.push_back(candlestick);

	// Iterate from the second element
	for (auto it = std::next(yearToTempsMap.begin()); it != yearToTempsMap.end(); ++it) {
		const std::string& year = it->first; // Extract the year (key) as a string.
```

```
            const std::vector<double>& temps = it->second; // Extract the vector of temperatures
            // Calculate the mean, high, and low temperatures
            meanHighLow = getMeanHighLow(temps);
            open = close; // previous years close
            close = meanHighLow[0]; // current year's mean
            high = meanHighLow[1];
            low = meanHighLow[2];

            // Create a Candlestick object and add it to the candlesticks vector
            Candlestick candlestick(year, open, close, high, low);
            candlesticks.push_back(candlestick);
        }

        return candlesticks;
}


std::vector<ScatterPlotHighLow> Statistics::calculateScatterPlotHighLows(const std::map<std::string,
        std::vector<double>>& yearToTempsMap)
{
        // Initialize an empty vector to store scatter plot data points.
        std::vector<ScatterPlotHighLow> scatterPlot;

        // If the input map is empty, return the empty scatter plot vector.
        if (yearToTempsMap.empty()) {
            return scatterPlot;
        }

        // Iterate through each entry in the map.
        for (const auto& pair : yearToTempsMap) {
            const std::string& year = pair.first; // Extract the year (key) as a string.
            const std::vector<double>& temps = pair.second; // Extract the vector of temperatures
            // Calculate the high and low temperatures
            std::pair<double, double> highLow = getHighLow(temps);
            double high = highLow.first;
            double low = highLow.second;

            // Create a ScatterPlotHighLow object and add it to the ScatterPlot vector
            ScatterPlotHighLow ScatterPlotHighLow(year, high, low);
            scatterPlot.push_back(ScatterPlotHighLow);
        }
```

```
	return scatterPlot;
;
}



/////////////// Private methods /////////////////

std::vector<double> Statistics::getMeanHighLow(const std::vector<double>& temps)
{
	// Check if the input vector is empty. If so, return a default vector of zeros.
	if (temps.empty()) {
		return { 0.0, 0.0, 0.0 };
	}

	// Initialize variables to calculate the sum, high, and low values.
	double sum = 0.0;		// To accumulate the total sum of temperatures.
	double high = temps[0];		// Set the initial high value to the first temperature.
	double low = temps[0];		// Set the initial low value to the first temperature.

	// Loop through each temperature in the vector.
	for (double t : temps) {
		sum += t; // Add the current temperature to the sum.
		if (t > high) // Update the high value if the current temperature is greater.
			high = t;
		if (t < low) // Update the low value if the current temperature is smaller.
			low = t;
	}

	// Return a vector containing the mean, high, and low values.
	return { sum / temps.size(), high, low };
}


std::pair<double, double> Statistics::getHighLow(const std::vector<double>& temps)
{
	// Check if the input vector is empty. If so, return a default pair of zeros.
	if (temps.empty()) {
		return { 0.0, 0.0 };
	}

	// Initialize variables to calculate the high and low values.
```

```
	double high = temps[0];		// Set the initial high value to the first temperature.
	double low = temps[0];		// Set the initial low value to the first temperature.

	// Loop through each temperature in the vector.
	for (double t : temps) {
		if (t > high) // Update the high value if the current temperature is greater.
			high = t;
		if (t < low) // Update the low value if the current temperature is smaller.
			low = t;
	}

	// Return a pair containing the high and low values.
	return { high, low };
}



double Statistics::getCorrelationCoefficient(std::vector<std::pair<std::string, double>> predictionData)
{
	// Check if the input data is empty and throw an exception if it is.
	if (predictionData.empty()) {
		throw std::invalid_argument("Prediction data cannot be empty.");
	}

	// Initialize variables to compute the sums for the correlation formula.
	double sumX = 0.0, sumY = 0.0, sumXY = 0.0;
	double sumXSquared = 0.0, sumYSquared = 0.0;
	// Iterate through each pair in the input vector -
	// Pair.first: year or day (x-value)
	// Pair.second: temperature (y-value)
	for (const auto& pair : predictionData) {
		double X = std::stod(pair.first);
		double Y = pair.second;
		// Accumulate the sums needed for the correlation coefficient formula.
		sumX += X;			// Sum of X values.
		sumY += Y;			// Sum of Y values.
		sumXY += X * Y;			// Sum of the product of X and Y values.
		sumXSquared += pow(X, 2);	// Sum of squared X values.
		sumYSquared += pow(Y, 2);	// Sum of squared Y values.
	}

	// Number of data points in the input vector.
```

```
	size_t numDataPoints = predictionData.size();
	// Calculate the Pearson correlation coefficient (r) using the formula.
	double r = (numDataPoints * sumXY - sumX * sumY) /
		sqrt((numDataPoints * sumXSquared - pow(sumX, 2)) *
			(numDataPoints * sumYSquared - pow(sumY, 2)));

	return r;
}


double Statistics::getLinearRegressionPrediction(std::vector<std::pair<std::string, double>>
predictionData,
	double year)
{
	// Check if the input data is empty and throw an exception if it is.
	if (predictionData.empty()) {
		throw std::invalid_argument("Prediction data cannot be empty.");
	}

	// Initialize variables to compute the necessary sums for simple linear regression.
	double sumX = 0.0, sumY = 0.0, sumXY = 0.0;
	double sumXSquared = 0.0;
	// Iterate through each pair in the input vector -
	// Pair.first: year or day (x-value)
	// Pair.second: temperature (y-value)
	for (const auto& pair : predictionData) {
		double X = std::stod(pair.first);
		double Y = pair.second;
		// Accumulate sums needed for slope and intercept calculations.
		sumX += X;			// Sum of all X values.
		sumY += Y;			// Sum of all Y values.
		sumXY += X * Y;			// Sum of the product of X and Y.
		sumXSquared += pow(X, 2);	// Sum of squared X values.
	}

	// Number of data points in the input vector.
	size_t numDataPoints = predictionData.size();

	// Compute the numerator and denominator for the slope (m) of the regression line.
	double mNumerator = (numDataPoints * sumXY - sumX * sumY);
	double mDenominator = numDataPoints * sumXSquared - pow(sumX, 2);
	// Check for a zero denominator, which indicates the inability to calculate a regression line.
```

```
	if (mDenominator == 0.0) {
		throw std::runtime_error("Cannot calculate linear regression");
	}
	// Calculate the slope (m) of the regression line.
	double m = mNumerator / mDenominator;
	// Calculate the y-intercept (b) of the regression line.
	double b = (sumY - m * sumX) / numDataPoints;

	// Predict the dependent variable (Y) for the given independent variable (year).
	return m * year + b;
}
```

ScatterPlotHighLow.h

```
#pragma once

#include <string>
#include <map>
#include <vector>


class ScatterPlotHighLow
{
public:
	ScatterPlotHighLow(std::string _year, double _high, double _low);

	// Member getters
	const std::string& getYear() const;
	double getHigh() const;
	double getLow() const;

	// Static method: Calculates the scatter plot data for highs.
	// Takes a vector of ScatterPlotHighLow objects and maps the Y-axis values to their graphical representation.
	static std::map<int, std::string, std::greater<int>> calculateScatterPlotHighs(
		const std::vector<ScatterPlotHighLow>& scatterPlotHighLows);
	// Static method: Calculates the scatter plot data for lows.
```

```
	// Takes a vector of ScatterPlotHighLow objects and maps the Y-axis values to their graphical representation.
	static std::map<int, std::string, std::greater<int>> calculateScatterPlotLows(
		const std::vector<ScatterPlotHighLow>& scatterPlotHighLows);

private:
	const std::string year;
	double high;
	double low;

	// Static helper method: Calculates the Y-axis values for high points in the scatter plot.
	// Returns a map where:
	// - The key is the y axis
	// - The value is an empty string
	static std::map<int, std::string, std::greater<int>> calculateYAxisHighs(const std::vector<ScatterPlotHighLow>& scatterPlotHighLows);
	// Static helper method: Calculates the Y-axis values for low points in the scatter plot.
	// Returns a map where:
	// - The key is the y axis
	// - The value is an empty string
	static std::map<int, std::string, std::greater<int>> calculateYAxisLows(const std::vector<ScatterPlotHighLow>& scatterPlotHighLows);
};
```

ScatterPlotHighLow.cpp

```
#include "ScatterPlotHighLow.h"

#include <cmath>


ScatterPlotHighLow::ScatterPlotHighLow(std::string _year, double _high, double _low)
	: year(_year), high(_high), low(_low)
{

}


const std::string& ScatterPlotHighLow::getYear() const {
```

```
	return year;
}


double ScatterPlotHighLow::getHigh() const {
	return high;
}

double ScatterPlotHighLow::getLow() const {
	return low;
}



std::map<int, std::string, std::greater<int>> ScatterPlotHighLow::calculateScatterPlotHighs(
	const std::vector<ScatterPlotHighLow>& scatterPlotHighLows)
{
	// Calculate the Y-axis scale for the candlestick chart.
	// The Y-axis values are stored in a map ordered in descending order (std::greater<int>).
	std::map<int, std::string, std::greater<int>> scatterPlot =
calculateYAxisHighs(scatterPlotHighLows);

	// ANSI escape codes for color formatting (green for high points).
	const std::string reset = "\033[0m";
	const std::string green = "\033[32m";

	// Iterate over the vector of ScatterPlotHighLow objects.
	for (const auto& p : scatterPlotHighLows) {
		// Round the high value to the nearest integer for plotting.
		int high = static_cast<int>(std::round(p.getHigh()));

		// Iterate over the scatter plot map.
		for (auto& pair : scatterPlot) {
			// If the current Y-axis level matches the high value, add a green "+" symbol.
			if (pair.first == high)
				pair.second += green + "+" + "  " + reset;
			// Else add 3 empty spaces to fill the space to the next year
			else
				pair.second += "   ";
		}

	}
```

```
	return scatterPlot;
}


std::map<int, std::string, std::greater<int>> ScatterPlotHighLow::calculateScatterPlotLows(
	const std::vector<ScatterPlotHighLow>& scatterPlotHighLows)
{
	// Calculate the Y-axis scale for the candlestick chart.
	// The Y-axis values are stored in a map ordered in descending order (std::greater<int>).
	std::map<int, std::string, std::greater<int>> scatterPlot =
calculateYAxisLows(scatterPlotHighLows);
	// ANSI escape codes for color formatting (red for low points).
	const std::string reset = "\033[0m";
	const std::string red = "\033[31m";

	// Iterate over the vector of ScatterPlotHighLow objects.
	for (const auto& p : scatterPlotHighLows) {
		int low = static_cast<int>(std::round(p.getLow()));
		// Iterate over the scatter plot map.
		for (auto& pair : scatterPlot) {
			// If the current Y-axis level matches the low value, add a red "+" symbol.
			if (pair.first == low)
				pair.second += red + "+" + "  " + reset;
			// Else add 3 empty spaces to fill the space to the next year
			else
				pair.second += "   ";
		}
	}
	return scatterPlot;
}


std::map<int, std::string, std::greater<int>> ScatterPlotHighLow::calculateYAxisHighs(const
std::vector<ScatterPlotHighLow>& scatterPlotHighLows)
{
	// calculate y-axis scale
	// The Y-axis values are stored in a map ordered in descending order (std::greater<int>).
	std::map<int, std::string, std::greater<int>> yAxis;

	// Start the min and max high values with the first high value
	double yAxisMax = scatterPlotHighLows[0].getHigh();
	double yAxisMin = scatterPlotHighLows[0].getHigh();
```

```
	// Iterate over the vector of ScatterPlotHighLow objects.
	for (const auto& p : scatterPlotHighLows) {
		// Update the maximum and minimum Y-axis values if necessary.
		if (p.getHigh() > yAxisMax)
			yAxisMax = p.getHigh();
		if (p.getHigh() < yAxisMin)
			yAxisMin = p.getHigh();
	}
	// Round the maximum and minimum Y-axis values to the nearest integers to scale the y-axis
	// and allow data points to be rounded and assigned to a key in the map
	int yAxisMaxRound = static_cast<int>(std::round(yAxisMax));
	int yAxisMinRound = static_cast<int>(std::round(yAxisMin));

	// Insert the map keys from the max high to the min high and assign an empty string as each
value.
	for (int i = yAxisMaxRound; i >= yAxisMinRound; i -= 1) {
		yAxis[i] = "";
	}

	return yAxis;
}


std::map<int, std::string, std::greater<int>> ScatterPlotHighLow::calculateYAxisLows(const
std::vector<ScatterPlotHighLow>& scatterPlotHighLows)
{
	// calculate y-axis scale
	// The Y-axis values are stored in a map ordered in descending order (std::greater<int>).
	std::map<int, std::string, std::greater<int>> yAxis;
	// Start the min and max high values with the first low value
	double yAxisMax = scatterPlotHighLows[0].getLow();
	double yAxisMin = scatterPlotHighLows[0].getLow();

	// Iterate over the vector of ScatterPlotHighLow objects.
	for (const auto& p : scatterPlotHighLows) {
		// Update the maximum and minimum Y-axis values if necessary.
		if (p.getLow() > yAxisMax)
			yAxisMax = p.getLow();
		if (p.getLow() < yAxisMin)
			yAxisMin = p.getLow();
	}
```

```
        // Round the maximum and minimum Y-axis values to the nearest integers to scale the y-axis
        // and allow data points to be rounded and assigned to a key in the map.
        int yAxisMaxRound = static_cast<int>(std::round(yAxisMax));
        int yAxisMinRound = static_cast<int>(std::round(yAxisMin));

        // Insert the map keys from the max low to the min low and assign an empty string as each
value.
        for (int i = yAxisMaxRound; i >= yAxisMinRound; i -= 1) {
                yAxis[i] = "";
        }

        return yAxis;
}
```

WeatherAnalyzerMain.h

```
// Ensure this header file is included only once
#pragma once

// Project headers
#include "TemperatureRow.h"
#include "csvReader.h"
#include "Candlestick.h"
#include "ScatterPlotHighLow.h"


class WeatherAnalyzerMain
{
public:
   WeatherAnalyzerMain();
   void init();

   // Access is only required within the class for the following data members and methods.
private:
   // Stores user-selected country
   std::string country = "";
   // Stores the user-selected time period filter.
   std::string period = "";
```

```
// Stores the user-selected month.
std::string month = "";
// Stores the user-selected day.
std::string day = "";

////////// Methods called by the user from the Menu ///////////

// Prints information about the dataset
void about();

// Prints a temperature from an exact date and time.
void getTemperature();

// Prints candlestick data for a time period.
void printCandlestickData();

// Overloaded method.
// Plots a candlestick chart using the printed data from the method above
void plotCandlestickChart(std::vector<Candlestick> candlesticks);

// Overloaded method.
// Plots a candlestick chart without the user printing the candlestick data first
void plotCandlestickChart();

// Prints scatter plot data for a time period.
void printScatterPlotData();

// Plots a scatter plot.
void plotScatterPlot();

// Calculates and displays temperature predictions.
void getPredictionTemp();

////////////////////////////////////////////////////////////


// Prints the main menu to the user.
void printMenu();

// Calls a method from the user-selected menu option. Returns a user option that is saved globally.
std::string processOption();
```

```
// Stores the currently selected menu option from the processOption method.
std::string option;

// Calculates candlestick data. Returns a vector of Candlestick objects.
std::vector<Candlestick> getCandlestickData();

// Calculates scatter plot data. Returns a vector of ScatterPlotHighLow objects.
std::vector<ScatterPlotHighLow> getScatterPlotData();

//////////////// Helper Methods //////////////////

// Gets user inputs for country, period, month, and day.
void getUserInput();

// Gets user input for country.
std::string getUserCountry();

// Gets user input for period filter (e.g., year or day of the year).
std::string getUserPeriodFilter();

// Asks user if they would like to view a candlestick chart for the menu option 3.
std::string getUserCandlestickOption();

// Gets user input for a year range.
std::pair<std::string, std::string> getUserYearRange();

// Asks user if they would like to view high or low data.
std::string getUserHighLow();

// Gets user input for year.
std::string getUserYear();

// Gets user input for month.
std::string getUserMonth();

// Gets user input for day.
std::string getUserDay();

// Gets user input for hour.
std::string getUserHour();

// Asks the user which year they would like to predict data for.
```

```
    double getUserPredictionYear();

    // Map linking menu options to corresponding member function pointers.
    std::map<std::string, void(WeatherAnalyzerMain::*)()> OPTIONS;

    // Stores rows of temperature data.
    std::vector<TemperatureRow> rows;


    // Maps month numbers to month names.
    std::map<std::string, std::string> months = {
        {"01", "January"}, {"02", "February"}, {"03", "March"}, {"04", "April"},
        {"05", "May"}, {"06", "June"}, {"07", "July"}, {"08", "August"},
        {"09", "September"}, {"10", "October"}, {"11", "November"}, {"12", "December"}
    };
};
```

WeatherAnalyzerMain.cpp

```
// Related header
#include "WeatherAnalyzerMain.h"

// Standard library includes
#include <iomanip>
#include <thread>

// Project headers
#include "Statistics.h"
#include "ScatterPlotHighLow.h"
#include "Candlestick.h"


// Initialize the OPTIONS map with menu options the user can select as keys
// and pointers to the member function that handles the option as values.
WeatherAnalyzerMain::WeatherAnalyzerMain()
{
    OPTIONS = {
```

```
        {"1", &WeatherAnalyzerMain::about},
        {"2", &WeatherAnalyzerMain::getTemperature},
        {"3", &WeatherAnalyzerMain::printCandlestickData},
        {"4", &WeatherAnalyzerMain::plotCandlestickChart},
        {"5", &WeatherAnalyzerMain::printScatterPlotData},
        {"6", &WeatherAnalyzerMain::plotScatterPlot},
        {"7", &WeatherAnalyzerMain::getPredictionTemp}
    };
}


void WeatherAnalyzerMain::init()
{
    try
    {
        // Attempt to read the csv file and create a TemperatureRow type for each row, store the rows in a
vector
        rows = CsvReader::readcsv("weather_data_EU_1980-2019_temp_only.csv");
    }
    catch (const std::exception& e)
    {
        // Catch any exception thrown during the CSV file reading process and print an
        // error message to notify the user about the failure.
        std::cout << "Exception caught: " << e.what() << std::endl;
    }

    // Display the menu and process user inputs in a loop until the user chooses to exit.
    do {
        // Display the menu options to the user.
        printMenu();
        // Process the user's input and update the `option` member variable with the selected option.
        WeatherAnalyzerMain::option = processOption();

    } while (WeatherAnalyzerMain::option != "e"); // Exit the loop if the user selects "e" for exit.
    // Notify the user that the application is exiting the menu and closing.
    std::cout << "Exiting options" << std::endl;
}


void WeatherAnalyzerMain::printMenu()
{
    std::vector<std::string> MENU = {
```

```
        "\n======================",
        "Please enter an option 1-6 or press e to exit",
        "======================",
        "1: About",
        "2: Get a Temperature",
        "3: Print Candlestick Data",
        "4: View Candlestick Chart",
        "5: Print Scatter Plot Data",
        "6: View Scatter Plot",
        "7: Predict a high or low",
        "======================"
    };

    // Iterate through the menu options and print each one to the console.
    for (const auto& option : MENU) {
        std::cout << option << std::endl;
    }
}


std::string WeatherAnalyzerMain::processOption()
{
    std::string option;
    std::getline(std::cin, option);
    // Check if the input matches any valid options in the OPTIONS map.
    if (OPTIONS.find(option) != OPTIONS.end()) {
        std::cout << "You chose option " << option << ".\n" << std::endl; // print the user option
        (this->*OPTIONS[option])(); // dereference the pointer to the member function and call it on the
current instance
    }
    // Handle invalid entries unless the user chooses to exit.
    else if (option != "e") {
        std::cout << "Invalid entry" << std::endl;
    }
    return option;
}


void WeatherAnalyzerMain::about()
{
    std::cout << "---About Section---" << std::endl;
```

```
   std::cout << "The dataset contains an hourly temperature in Degrees Celcius from 1980 to 2019" <<
std::endl;
   std::cout << "for 28 European countries. To determine the temperature, it uses a" << std::endl;
   std::cout << "population-weighted mean across all MERRA-2 grid cells within each country." <<
std::endl;
   std::cout << "The original dataset can be found here:
https://data.open-power-system-data.org/weather_data/2020-09-16." << std::endl;
}


void WeatherAnalyzerMain::getTemperature()
{
   // call helper function to get user inputs.
   std::string country = getUserCountry();
   std::string year = getUserYear();
   std::string month = getUserMonth();
   std::string day = getUserDay();
   std::string hour = getUserHour();

   // Build a timestamp with the user inputs.
   std::string timestamp = year + "-" + month + "-" + day + "T" + hour + ":00:00Z";

   try
   {
      double temp;
      unsigned int rowIndex = TemperatureRow::getRowIndex(rows, timestamp); // Get the row index for
the timestamp.
      unsigned int tempIndex = TemperatureRow::countries.at(country); // Get the column index for the
country.
      temp = rows[rowIndex].TemperatureRow::getTemperatures()[tempIndex]; // Get the temperature
      std::cout << "\nCountry: " << country
            << "\nDate: " << day << " " << months[month] << " " << year
            << "\nTime: " << hour << ":00"
            << "\nTemperature Degrees Celcius: " << temp
            << std::endl;

   }
   catch (const std::exception& e)
   {
      std::cout << "No data" << e.what() << std::endl;
   }
}
```

```
// get only the data for the candlesticks
std::vector<Candlestick> WeatherAnalyzerMain::getCandlestickData()
{
   getUserInput();
   // Get the country index if found.
   unsigned int countryIndex = 0;
   auto it = TemperatureRow::countries.find(country);
   if (it != TemperatureRow::countries.end()) {
      countryIndex = it->second;
   }
   else {
      std::cerr << "Error: No country at that index." << country << std::endl;
      throw std::runtime_error("Invalid country input.");
   }

   std::map<std::string, std::vector<double>> yearToTempsMap;
   std::vector<Candlestick> candlesticks;

   // If the period is "1" (Year), calculate candlesticks for each year.
   if (period == "1") {
      std::cout << "You chose Year" << std::endl;
      yearToTempsMap = TemperatureRow::getTempsByYear(rows, countryIndex); // Filter the data
      candlesticks = Statistics::calculateCandlesticks(yearToTempsMap); // Calculate each candlestick
   }
   // If the period is "2" (Day of the year), calculate candlesticks for a day of each year.
   else {
      std::cout << "You chose Day" << std::endl;
      std::string monthDay = month + "-" + day;
      yearToTempsMap = TemperatureRow::getTempsByDayOfYear(rows, countryIndex, monthDay); //
Filter the data
      candlesticks = Statistics::calculateCandlesticks(yearToTempsMap); // Calculate each candlestick
   }

   return candlesticks;
}


void WeatherAnalyzerMain::printCandlestickData()
{
   // Call the above method to get the data.
   std::vector<Candlestick> candlesticks = getCandlestickData();
```

```
    // Print the candlestick data with 3 decimal places.
    std::cout << "YEAR |  OPEN |  CLOSE |  HIGH  |  LOW" << std::endl;
    std::cout << "--------------------------------------" << std::endl;
    for (const auto& c : candlesticks) {
        std::cout << std::fixed << std::setprecision(3) <<
            c.getYear() << " | " << c.getOpen() << " | " << c.getClose() <<
            " | " << c.getHigh() << " | " <<c.getLow() << std::endl;
    }

    // ask user if they want to plot the candlesticks
    std::string answer = getUserCandlestickOption();

    if (answer == "1") {
        std::cout << "You chose Yes" << std::endl;
        plotCandlestickChart(candlesticks);
    }
    else {
        std::cout << "You chose No" << std::endl;
    }
}


void WeatherAnalyzerMain::plotCandlestickChart(std::vector<Candlestick> candlesticks)
{
    std::map<int, std::string, std::greater<int>> chart = Candlestick::getCandlestickChart(candlesticks);
    // Determine the range of years to be displayed on the x-axis based on the candlestick data.
    // Use the same method as the overloaded function below to avoid changing code.
    std::string yearStart = candlesticks[0].getYear(); // Get the first year
    std::string yearEnd = candlesticks.back().getYear(); // Get the last year
    // Create an iterator to print the x axis years (in this function there is no range but
    // leaving it like this allows to easily add year range functionality)
    auto startIt = TemperatureRow::years.lower_bound(yearStart);
    auto endIt = TemperatureRow::years.upper_bound(yearEnd);

    std::cout << std::endl;
    // Iterate over the map and print each y-axis line of candlestick data for every year
    for (const auto& pair : chart) {
        std::cout << std::setw(4)
            << pair.first << "   "
            << pair.second
            << std::endl;
```

```
        std::this_thread::sleep_for(std::chrono::milliseconds(50)); // Slow the printing for effect
    }

    // Print the x axis years - used vertical to align with the candles
    std::string xAxisTop = "       ";
    std::string xAxisBottom = "       ";
    for (auto it = startIt; it != endIt; ++it) {
        xAxisTop += it->first.substr(2, 1) + "  ";
        xAxisBottom += it->first.substr(3, 1) + "  ";
    }
    std::cout << "\n" << xAxisTop << std::endl;
    std::cout << xAxisBottom << std::endl;
}


void WeatherAnalyzerMain::plotCandlestickChart()
{
    std::vector<Candlestick> candlesticks = getCandlestickData();
    // Get a user year range to filter the data by year
    std::pair<std::string, std::string> yearRange = getUserYearRange();
    std::string yearStart = yearRange.first;
    std::string yearEnd = yearRange.second;
    // Get the index for the start and end years in the candlesticks vector
    unsigned int yearStartIndex = TemperatureRow::years.at(yearStart);
    unsigned int yearEndIndex = TemperatureRow::years.at(yearEnd);

    // Create a new (filtered by year range) vector of candlesticks
    std::vector<Candlestick> filteredCandlesticks(
        candlesticks.begin() + yearStartIndex,
        candlesticks.begin() + yearEndIndex + 1);

    // Call the above overloaded method of the same name with the filtered candlesticks
    plotCandlestickChart(filteredCandlesticks);
}


std::vector<ScatterPlotHighLow> WeatherAnalyzerMain::getScatterPlotData()
{
    getUserInput();
    // Get the country index if found.
    unsigned int countryIndex = 0;
    auto it = TemperatureRow::countries.find(country);
```

```
    if (it != TemperatureRow::countries.end()) {
        countryIndex = it->second;
    }
    else {
        std::cerr << "Error: Invalid country entered: " << country << std::endl;
        throw std::runtime_error("Invalid country input.");
    }
    std::map<std::string, std::vector<double>> yearToTempsMap;
    std::vector<ScatterPlotHighLow> scatterPlot;

    // If the period is "1" (Year), calculate ScatterPlotHighLow for each year.
    if (period == "1") {
        yearToTempsMap = TemperatureRow::getTempsByYear(rows, countryIndex);
        scatterPlot = Statistics::calculateScatterPlotHighLows(yearToTempsMap);
    }
    // If the period is "2" (Day of the year), calculate ScatterPlotHighLow for a day of each year.
    else {
        std::string monthDay = month + "-" + day;
        yearToTempsMap = TemperatureRow::getTempsByDayOfYear(rows, countryIndex, monthDay);
        scatterPlot = Statistics::calculateScatterPlotHighLows(yearToTempsMap);
    }

    return scatterPlot;
}


void WeatherAnalyzerMain::printScatterPlotData()
{
    std::vector<ScatterPlotHighLow> scatterPlot = getScatterPlotData();

    // Print the ScatterPlotHighLow data with 3 decimal places.
    std::cout << "YEAR |  HIGH  |  LOW" << std::endl;
    std::cout << "----------------------" << std::endl;
    for (const auto& p : scatterPlot) {
        std::cout << std::fixed << std::setprecision(3) <<
            p.getYear() << " | " << p.getHigh() << " | " << p.getLow() << std::endl;
    }
}


void WeatherAnalyzerMain::plotScatterPlot()
{
```

```
std::vector<ScatterPlotHighLow> scatterPlot = getScatterPlotData();
// Get a user year range to filter the data by year
std::pair<std::string, std::string> yearRange = getUserYearRange();
std::string yearStart = yearRange.first;
std::string yearEnd = yearRange.second;
// Get the index for the start and end years in the candlesticks vector
unsigned int yearStartIndex = TemperatureRow::years.at(yearStart);
unsigned int yearEndIndex = TemperatureRow::years.at(yearEnd);

// Create a new (filtered by year range) vector of ScatterPlotHighLow objects
std::vector<ScatterPlotHighLow> filteredScatterPlot(
   scatterPlot.begin() + yearStartIndex,
   scatterPlot.begin() + yearEndIndex + 1);

// Create a map to store the plot
std::map<int, std::string, std::greater<int>> plot;
// Ask the user if they want high or low plotted.
std::string highOrLow = getUserHighLow();
if (highOrLow == "High") {
   // Calculate the plot with high values for the year range
   plot = ScatterPlotHighLow::calculateScatterPlotHighs(filteredScatterPlot);
}
else
   // Calculate the plot with low values for the year range
   plot = ScatterPlotHighLow::calculateScatterPlotLows(filteredScatterPlot);

// Create an iterator to print the x axis years
auto startIt = TemperatureRow::years.lower_bound(yearStart);
auto endIt = TemperatureRow::years.upper_bound(yearEnd);

std::cout << std::endl;
// Iterate over the map and print each y-axis line of scatter plot data for every year
for (const auto& pair : plot) {
   std::cout << std::setw(4)
      << pair.first << "    "
      << pair.second
      << std::endl;
   std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
// Print the x axis years - used vertical to align with the candles
std::string xAxisTop = "       ";
std::string xAxisBottom = "       ";
```

```
    for (auto it = startIt; it != endIt; ++it) {
        xAxisTop += it->first.substr(2, 1) + "  ";
        xAxisBottom += it->first.substr(3, 1) + "  ";
    }
    std::cout << "\n" << xAxisTop << std::endl;
    std::cout << xAxisBottom << std::endl;
}


void WeatherAnalyzerMain::getPredictionTemp()
{
    // Get scatter plot data for the selected country and period.
    std::vector<ScatterPlotHighLow> scatterPlot = getScatterPlotData();
    // Ask the user if they want to predict high or low temperatures.
    std::string extremumTemp = getUserHighLow();
    // Vector to store the prediction data (year and corresponding high/low).
    std::vector<std::pair<std::string, double>> predicationData;
    // Pushes year and either high or low data as a pair to the vector
    if (extremumTemp == "High") {
        for (const auto p : scatterPlot) {
            predicationData.push_back({ p.getYear(), p.getHigh()} );
        }
    }
    else {
        for (const auto p : scatterPlot) {
            predicationData.push_back({ p.getYear(), p.getLow() });
        }
    }

    // Calculates the correlation coefficient data
    double r = Statistics::getCorrelationCoefficient(predicationData);
    // if r is stronger than 0.7 or -0.7 then use simple linear regression to make the prediction
    if (r >= 0.7 || r <= -0.7) {
        std::cout << "The correlation coefficient (r) is strong: " << r << std::endl;
        std::cout << "(Prediction calculated using simple linear regression)." << std::endl;
        // Ask the user for a year to predict
        double predYear = getUserPredictionYear();
        // Predict the temperature using linear regression and round the value
        int predTemp =
static_cast<int>(std::round(Statistics::getLinearRegressionPrediction(predicationData, predYear)));
        // The user choose yearly data
        if (period == "1") {
```

```
            std::cout << "\nCountry: " << country
                << "\n" << extremumTemp << " Temperature Degrees Celcius : " << predTemp
                << std::endl;
        }
        // The user chose a day of the year
        else {
            std::cout << "\nCountry: " << country
                << "\nDate: " << day << " " << months[month]
                << "\n" << extremumTemp << " Temperature Degrees Celcius : " << predTemp
                << std::endl;
        }
    }
    // If r is weaker make the prediction using the average from the previous years
    else {
        // average prediction
        std::cout << "The correlation coefficient (r) is weak: " << r << std::endl;
        std::cout << "(Prediction calculated using historical mean)." << std::endl;
        // Calculate the average
        double sum = 0.0;
        for (const auto& pair : predicationData) {
            sum += pair.second;
        }
        int average = static_cast<int>(std::round(sum / predicationData.size()));
        // The user choose yearly data
        if (period == "1") {
            std::cout << "\nCountry: " << country
                << "\n" << extremumTemp << " Temperature Degrees Celcius : " << average
                << std::endl;
        }
        // The user chose a day of the year
        else {
            std::cout << "\n---Yearly mean prediction---"
                << "\nCountry: " << country
                << "\nDate: " << day << " " << months[month]
                << "\n" << extremumTemp << " Temperature Degrees Celcius : " << average
                << std::endl;
        }

    }

}
```

```
////////////////// helper methods ///////////////////////

void WeatherAnalyzerMain::getUserInput()
{
   country = getUserCountry();
   period = getUserPeriodFilter();

   if (period != "1") {
      month = getUserMonth();
      day = getUserDay();
   }
}


std::string WeatherAnalyzerMain::getUserCountry()
{
   std::string country;
   do {
      std::cout << "---Input instructions---\n" << std::endl;
      std::cout << "Choose a country from the following selection:" << std::endl;
      for (const auto& pair : TemperatureRow::countries) {
         std::cout << pair.first << std::endl;
      }
      std::cout << "-----------------" << std::endl;
      std::getline(std::cin, country);

      for (auto& u : country) {
         u = std::toupper(u);
      }

   } while (TemperatureRow::countries.find(country) == TemperatureRow::countries.end());

   std::cout << "You chose: " << country << std::endl;
   return country;
}


std::string WeatherAnalyzerMain::getUserPeriodFilter()
{
   std::string period;
   do {
      std::cout << "---Input instructions---" << std::endl;
```

```
        std::cout << "Choose period of data to be Year or Day" << std::endl;
        std::cout << "Please enter an option 1-2" << std::endl;
        std::cout << "======================" << std::endl;
        std::cout << "1: Year" << std::endl;
        std::cout << "2: Day" << std::endl;
        std::cout << "======================" << std::endl;

        std::getline(std::cin, period);
    } while (period != "1" && period != "2");

    return period;
}


std::string WeatherAnalyzerMain::getUserCandlestickOption()
{
    std::string answer;
    do {
        std::cout << std::endl;
        std::cout << "Would you like to plot the candlesticks?" << std::endl;
        std::cout << "======================" << std::endl;
        std::cout << "1: Yes" << std::endl;
        std::cout << "2: No" << std::endl;
        std::cout << "======================" << std::endl;

        std::getline(std::cin, answer);
    } while (answer != "1" && answer != "2");

    return answer;
}


std::pair<std::string, std::string> WeatherAnalyzerMain::getUserYearRange()
{
    std::string yearStart;
    std::string yearEnd;

    do {
        std::cout << "Choose a year range between 1980-2019 (inclusive)." << std::endl;
        std::cout << "Enter start year:" << std::endl;
        std::getline(std::cin, yearStart);
        std::cout << "Enter end year:" << std::endl;
```

```
        std::getline(std::cin, yearEnd);
    } while (TemperatureRow::years.find(yearStart) == TemperatureRow::years.end() ||
            TemperatureRow::years.find(yearEnd) == TemperatureRow::years.end());

    return { yearStart, yearEnd };
}


std::string WeatherAnalyzerMain::getUserHighLow()
{
    std::string answer;
    do {
        std::cout << std::endl;
        std::cout << "Would you like the high or low?" << std::endl;
        std::cout << "======================" << std::endl;
        std::cout << "1: High" << std::endl;
        std::cout << "2: Low" << std::endl;
        std::cout << "======================" << std::endl;

        std::getline(std::cin, answer);
    } while (answer != "1" && answer != "2");

    if (answer == "1")
        return "High";
    else
        return "Low";
}


std::string WeatherAnalyzerMain::getUserYear()
{
    std::string year;
    do {
        std::cout << "Enter a year between 1980 and 2019:" << std::endl;
        std::getline(std::cin, year);
    } while (TemperatureRow::years.find(year) == TemperatureRow::years.end());

    return year;
}


std::string WeatherAnalyzerMain::getUserMonth()
```

```
{
   std::vector<std::string> months;
   for (int i = 1; i <= 12; ++i) {
      months.push_back((i < 10 ? "0" : "") + std::to_string(i));
   }
   std::string month;

   do {
      std::cout << "Enter a month between 01 and 12:" << std::endl;
      std::getline(std::cin, month);
   } while (std::find(months.begin(), months.end(), month) == months.end());

   return month;
}


std::string WeatherAnalyzerMain::getUserDay()
{
   std::vector<std::string> days;
   for (int i = 1; i <= 31; ++i) {
      days.push_back((i < 10 ? "0" : "") + std::to_string(i));
   }
   std::string day;

   do {
      std::cout << "Enter a day between 01 and 31:" << std::endl;
      std::getline(std::cin, day);
   } while (std::find(days.begin(), days.end(), day) == days.end());

   return day;
}


std::string WeatherAnalyzerMain::getUserHour()
{
   std::vector<std::string> hours;
   for (int i = 0; i <= 23; ++i) {
      hours.push_back((i < 10 ? "0" : "") + std::to_string(i));
   }
   std::string hour;

   do {
```

```
        std::cout << "Enter an hour between 00 and 23:" << std::endl;
        std::getline(std::cin, hour);
    } while (std::find(hours.begin(), hours.end(), hour) == hours.end());

    return hour;
}


double WeatherAnalyzerMain::getUserPredictionYear()
{
    std::string year;
    double yearNum = 0.0;
    do {
        std::cout << "What year would you like to predict?" << std::endl;
        std::getline(std::cin, year);
        yearNum = std::stod(year);
    } while (yearNum > 2019);

    return yearNum;
}
```