**File Overview:**

* **main.cpp:** The main program file that controls how everything runs.
* **CSVReader.h:** Definitions and instructions for reading CSV files.
* **CSVReader.cpp:** Code to read data from CSV files.
* **Candlestick.h:** Declares the functions for handling candlestick computation.
* **Candlestick.cpp:** Contains the implementation of functions of candlestick computation.
* **Weatherdata.h**: Declares functions for managing all the tasks of computing and plotting the graph.
* **Weatherdata.cpp**: Implementation of all the functions for all the tasks from task 1 to task 4.
* **merkelmain**.**h:** Declare the menu and display functions.
* **merkelmain.cpp**: Implement the menu and display functions and combine the implementation of the different tasks with the menu options.

**main.cpp**

// **Followed the starter code**

// Include the header file

#include "MerkelMain.h"

int main() {

    // An instance of the MerkelMain class

    MerkelMain app{};

    app.init();

}

**// End of following starter code**

**CSVReader.h**

// Include all the libraries

#pragma once

#include <vector>

#include <string>

#include <map>

#include "Candlestick.h"

class CSVReader {

public:

    CSVReader();

// **Followed the starter code**

    // Read the CSV file and return its contents as a vector of strings

    static std::vector<std::string> readCSV(const std::string& filePath);

    // Tokenize single line from the CSV file

    static std::vector<std::string> tokenise(const std::string &line, char seperator);

**// End of following starter code**

    // Get the index of a specific column in the CSV file

    static int Col\_Index(const std::string &col\_Name);

    // Group temperatures by year

    static std::map<int, std::vector<double>> group\_temp\_year(const std::vector<std::string>& dataset, int col\_Index);

    // Compute candlestick representation

    static std::vector<Candlestick> computeCandlesticks(const std::map<int, std::vector<double>>& yearly\_temps);

    // Group temperature data by year range

    static std::map<int, std::vector<double>> group\_temp\_year\_range(const std::vector<std::string>& dataset,int col\_Index,int startYear,int endYear);

    // process country codes and columns

    static std::map<std::string, int> CountryCode\_processing(const std::string& input);

};

**CSVReader.cpp**

// Include all the libraries and header file

#include "CSVReader.h"

#include "MerkelMain.h"

#include <fstream>

#include <iostream>

**// Followed starter code**

CSVReader::CSVReader()

{

}

// Reads a CSV file and returns its lines as a vector of strings

std::vector<std::string> CSVReader::readCSV(const std::string &filePath)

{

    std::ifstream file(filePath);

    // Vector to store each line from the csv

    std::vector<std::string> lines;

    // String to hold the current line

    std::string line;

    // If file is not open

    if (!file.is\_open())

    {

        std::cout << "Error: Could not open file: " << filePath << std::endl;

    }

    // Read the header row

    if (std::getline(file, line))

    {

        // Store the header row in the static member `headerRow` of MerkelMain

        MerkelMain::header\_Row = line;

    }

    // Read the lines and add them to the vector

    while (std::getline(file, line))

    {

        lines.push\_back(line);

    }

    // Close the file after reading

    file.close();

    return lines;

}

// Splits a string into tokens based on ","

std::vector<std::string> CSVReader::tokenise(const std::string &line, char seperator)

{

    // Vector to store the tokens

    std::vector<std::string> tokens;

    // Start of the current token

    int start = 0;

    // End of the current token

    int end = 0;

    // Loop to find each token

    while ((end = line.find(seperator, start)) != std::string::npos)

    {

        // Extract the token and add it to the vector

        tokens.push\_back(line.substr(start, end - start));

        start = end + 1;

    }

    tokens.push\_back(line.substr(start));

    return tokens;

}

**// End of following starter code**

// Finds the index in the CSV header row

int CSVReader::Col\_Index(const std::string &col\_Name)

{

    // Tokenise the header row to get individual column names

    std::vector<std::string> tokens = tokenise(MerkelMain::header\_Row, ',');

    for (int i = 0; i < tokens.size(); ++i)

    {

        // Compare the current token with the column name

        if (tokens[i] == col\_Name)

        {

            return static\_cast<int>(i);

        }

    }

    return -1;

}

// Function to group temperatures by year

std::map<int, std::vector<double>> CSVReader::group\_temp\_year(

    const std::vector<std::string>& dataset, int col\_Index) {

    // Store temperatures grouped by year

    std::map<int, std::vector<double>> yearly\_temps;

    for (const auto& line : dataset) {

        // Tokenise using ","

        auto tokens = CSVReader::tokenise(line, ',');

        if (tokens.size() <= col\_Index) continue;

        // Extract the year from the timpstamp

        int year = std::stoi(tokens[0].substr(0, 4));

        // Convert the temperature value to number

        double temperature = std::stod(tokens[col\_Index]);

        // Push the temperature to the particular year

        yearly\_temps[year].push\_back(temperature);

    }

    return yearly\_temps;

}

// Function to compute candlesticks

std::vector<Candlestick> CSVReader::computeCandlesticks(

    const std::map<int, std::vector<double>>& yearly\_temps) {

    // Store candlestick values

    std::vector<Candlestick> candlesticks;

    double prev\_close = 0.0;

    for (const auto& entry : yearly\_temps) {

        // Get the temperature for the current year

        const auto& temps = entry.second;

        // Compute candlestick for the current year using compute function in candlestick.cpp

        auto candle = Candlestick::compute(std::to\_string(entry.first), temps, prev\_close);

        // Add it to the vector

        candlesticks.emplace\_back(candle);

        prev\_close = candle.close;

    }

    return candlesticks;

}

// Function to group temperature by year range

std::map<int, std::vector<double>> CSVReader::group\_temp\_year\_range(const std::vector<std::string>& dataset,int col\_Index,int startYear,int endYear) {

    // Store temperature grouped by year within range

    std::map<int, std::vector<double>> yearly\_temps;

    for (const auto& line : dataset) {

        // tokenise using ","

        auto tokens = CSVReader::tokenise(line, ',');

        if (tokens.size() <= col\_Index) continue;

        // Get the timestamp

        std::string timestamp = tokens[0];

        // Get the year

        int year = std::stoi(timestamp.substr(0, 4));

        // Only compute for the specified year range

        if (year >= startYear && year <= endYear) {

            double temperature = std::stod(tokens[col\_Index]);

            yearly\_temps[year].push\_back(temperature);

        }

    }

    return yearly\_temps;

}

// Function to process the country code for the filter function

std::map<std::string, int> CSVReader::CountryCode\_processing(const std::string& input) {

    std::vector<std::string> code;

    std::map<std::string, int> code\_to\_index;

    std::string seperator = ",";

    int pos = 0;

    std::string input\_clean = input;

    // Extract and process country codes

    while ((pos = input\_clean.find(seperator)) != std::string::npos) {

        code.push\_back(input\_clean.substr(0, pos));

        input\_clean.erase(0, pos + seperator.length());

    }

    code.push\_back(input\_clean);

    for (std::string& countryCode : code) {

        // Trim spaces

        countryCode.erase(countryCode.find\_last\_not\_of(" \t\n\r\f\v") + 1);

        // Construct column name

        std::string col\_Name = countryCode + "\_temperature";

        int col\_Index = CSVReader::Col\_Index(col\_Name);

        // Country code not found print messages

        if (col\_Index == -1) {

            std::cout << "Error: Country code not found: " << countryCode << std::endl;

        } else {

            code\_to\_index[countryCode] = col\_Index;

        }

    }

    return code\_to\_index;

}

**Candlestick.h**

// Include all the required libraries

#pragma once

#include <string>

#include <vector>

class Candlestick {

public:

    // Candlestick variables

    // The year

    std::string date;

    // Opening temperature

    double open;

    // Highest temperature

    double high;

    // Lowest temperature

    double low;

    // Closing temperature

    double close;

    // Constructor to initialize a Candlestick object

    Candlestick(std::string \_date,

                double \_open,

                double \_high,

                double \_low,

                double \_close)

        : date(\_date), open(\_open), high(\_high), low(\_low), close(\_close) {}

    // Compute candlestick data

    static Candlestick compute(const std::string& date, const std::vector<double>& temperature, double prev\_close);

    // Text based graphs

    static void generateTextBasedGraph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode, bool isFuture);

    static void yearly\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode);

    static void Filter\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode);

    static void future\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode);

};

**Candlestick.cpp**

// Include the libraries and the header file

#include "Candlestick.h"

#include <iostream>

#include <algorithm>

#include <numeric>

#include <iomanip>

#include <limits>

#include <vector>

#include <cmath>

#include <string>

Candlestick Candlestick::compute(const std::string& date, const std::vector<double>& temperature, double prev\_close) {

    // Open value: Average temperature from the previous time frame

    double open = prev\_close;

    // High value: Highest temperature

    double high = \*std::max\_element(temperature.begin(), temperature.end());

    // Low value: Lowest temperature

    double low = \*std::min\_element(temperature.begin(), temperature.end());

    // Close value: Average temperature for the current time frame

    double close = std::accumulate(temperature.begin(), temperature.end(), 0.0) / temperature.size();

    return Candlestick(date, open, high, low, close);

};

void Candlestick::yearly\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode) {

    // Generate text-based candlestick chart

        double yMax = -std::numeric\_limits<double>::infinity();

        double yMin = std::numeric\_limits<double>::infinity();

        std::cout << "==================================================================================================================================" << std::endl;

        std::cout << "                                           Candlestick chart for country code: "  << countryCode << "\n";

        std::cout << "==================================================================================================================================\n" << std::endl;

        // y-axis range from candlesticks

        for (const auto &candle : candlesticks)

        {

            yMax = std::max(yMax, candle.high);

            yMin = std::min(yMin, candle.low);

        }

        // Normalize y-axis range for display

        int range = (static\_cast<int>(std::ceil(yMax)) + static\_cast<int>(std::floor(yMin))) / 2;

        int yMin\_Nor = range - 30;

        int yMax\_Nor = range + 30;

        // Generate the graph with colors

        for (int y = yMax\_Nor; y >= yMin\_Nor; y -= 1)

        {

            // y-axis label to be printed next to the temperature

            std::cout << std::setw(6) << y << " | ";

            for (const auto& candle : candlesticks)

            {

                // Round the values to display

                int H\_round = static\_cast<int>(std::round(candle.high));

                int L\_round = static\_cast<int>(std::round(candle.low));

                int O\_round = static\_cast<int>(std::round(candle.open));

                int C\_round = static\_cast<int>(std::round(candle.close));

                // Color based on Open (O) and Close (C)

                std::string color;

                if (C\_round > O\_round) {color = "\033[32m";} // Green for C > O, Red for O > C

                else {color = "\033[31m";}

                // Symbols for high,low,close,open

                if (y == H\_round) {std::cout << color << "H " << " \033[0m";}

                else if (y == L\_round) {std::cout << color << "L " << " \033[0m";}

                else if (y == O\_round) {std::cout << color << "O " << " \033[0m";}

                else if (y == C\_round) {std::cout << color << "C " << " \033[0m";}

                else if (y < H\_round && y > L\_round) {std::cout << color << "X " << " \033[0m";}

                else {std::cout << "   ";}

            }

            std::cout << std::endl;

        }

        std::cout << "-----------------------------------------------------------------------------------------------------------------------------------" << std::endl;

        // X-axis labels

        std::cout << "      ";

        for (int i = 0; i < candlesticks.size(); i++)

        {

            if (i % 10 == 0 || candlesticks[i].date == "1980" || candlesticks[i].date == "1990" || candlesticks[i].date == "2000" || candlesticks[i].date == "2019")

            {std::cout << std::setw(3) << candlesticks[i].date;}

            else{std::cout << std::setw(3) << "";}

        }

        std::cout << std::endl;

}

void Candlestick::Filter\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode) {

        // Generate text-based candlestick chart

        double yMax = -std::numeric\_limits<double>::infinity();

        double yMin = std::numeric\_limits<double>::infinity();

        for (const auto& candle : candlesticks) {

            yMax = std::max(yMax, candle.high);

            yMin = std::min(yMin, candle.low);

        }

        int range = (static\_cast<int>(std::ceil(yMax)) + static\_cast<int>(std::floor(yMin))) / 2;

        int yMin\_Nor = range - 30;

        int yMax\_Nor = range + 30;

        std::cout << "==================================================================================================================================" << std::endl;

        std::cout << "                                           Candlestick chart for country code: " << countryCode << "\n";

        std::cout << "==================================================================================================================================\n" << std::endl;

        for (int y = yMax\_Nor; y >= yMin\_Nor; y -= 1) {

            std::cout << std::setw(6) << y << " | ";

            for (const auto& candle : candlesticks) {

                int H\_round = static\_cast<int>(std::round(candle.high));

                int L\_round = static\_cast<int>(std::round(candle.low));

                int O\_round = static\_cast<int>(std::round(candle.open));

                int C\_round = static\_cast<int>(std::round(candle.close));

                // Color based on Open (O) and Close (C)

                std::string color;

                if (C\_round > O\_round) {color = "\033[32m";} // Green for C > O, Red for O > C

                else {color = "\033[31m";}

                if (y == H\_round) std::cout << color << "H" << "\033[0m\t";

                else if (y == L\_round) std::cout << color << "L" << "\033[0m\t";

                else if (y == O\_round) std::cout << color << "O" << "\033[0m\t";

                else if (y == C\_round) std::cout << color << "C" << "\033[0m\t";

                else if (y < H\_round && y > L\_round) std::cout << color << "X" << "\033[0m\t";

                else std::cout << " \t";

            }

            std::cout << std::endl;

        }

        std::cout << "---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------" << std::endl;

        std::cout << "  ";

        for (const auto& candle : candlesticks) {

            std::cout << std::setw(8) << candle.date;

        }

        std::cout << std::endl;

}

void Candlestick::future\_graph(const std::vector<Candlestick>& candlesticks, const std::string& countryCode) {

    double yMax = -std::numeric\_limits<double>::infinity();

    double yMin = std::numeric\_limits<double>::infinity();

    // Determine Y-axis range

    for (const auto& candle : candlesticks) {

        yMax = std::max(yMax, candle.high);

        yMin = std::min(yMin, candle.low);

    }

    int range = (static\_cast<int>(std::ceil(yMax)) + static\_cast<int>(std::floor(yMin))) / 2;

    int yMin\_Nor = range - 30;

    int yMax\_Nor = range + 30;

    std::cout << "\n==================================================================================================================================\n";

    std::cout << "                                           Candlestick Chart for Country Code: " << countryCode << "\n";

    std::cout << "==================================================================================================================================\n";

    for (int y = yMax\_Nor; y >= yMin\_Nor; y -= 1) {

        std::cout << std::setw(6) << y << " | ";

        for (const auto& candle : candlesticks) {

            int H\_round = static\_cast<int>(std::round(candle.high));

            int L\_round = static\_cast<int>(std::round(candle.low));

            int O\_round = static\_cast<int>(std::round(candle.open));

            int C\_round = static\_cast<int>(std::round(candle.close));

            // Determine the color for Open/Close

            std::string color;

            if (candle.date < "2020") {  // Historical data

                if (C\_round > O\_round) color = "\033[32m";  // Green for C > O

                else color = "\033[31m";  // Red for O > C

            } else {  // Future predictions

                if (C\_round > O\_round) color = "\033[34m";  // Blue for C > O

                else color = "\033[33m";  // Yellow for O > C

            }

            if (y == H\_round) std::cout << color << "H " << "\033[0m\t";

            else if (y == L\_round) std::cout << color << "L " << "\033[0m\t";

            else if (y == O\_round) std::cout << color << "O " << "\033[0m\t";

            else if (y == C\_round) std::cout << color << "C " << "\033[0m\t";

            else if (y < H\_round && y > L\_round) std::cout << color << "X " << "\033[0m\t";

            else std::cout << "  \t";

        }

        std::cout << std::endl;

    }

    // X-axis labels

    std::cout << "---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------\n";

    std::cout << "  ";

    for (const auto& candle : candlesticks) {

        std::cout << std::setw(8) << candle.date;

    }

    std::cout << std::endl;

}

**Weatherdata.h**

// Include all the libraries and the header file

#pragma once

#include "Candlestick.h"

class computation\_weather\_data{

    public:

        // Compute yearly candlestick data

        void compute\_candlestick\_yearly(const std::vector<std::string>& dataset);

        // Generate a text-based plot of yearly weather data

        void text\_plot\_yearly(const std::vector<std::string>& dataset);

        // Generate a text-based plot of filtered weather data based on the selected countries and year range

        void text\_plot\_Filters(const std::vector<std::string>& dataset);

        // Predict future weather data trends

        void Future\_Prediction(const std::vector<std::string>& dataset);

    private:

        // Store candlestick data as a vector

        std::vector<Candlestick> candlesticks;

};

**Weatherdata.cpp**

// Include function to add in the libraries and the header files needed to work on the functions

#include "weatherdata.h"

#include "MerkelMain.h"

#include "CSVReader.h"

#include "Candlestick.h"

#include <iostream>

#include <iomanip>

#include <algorithm>

#include <numeric>

#include <cmath>

**// TASK 1**

// Compute yearly candlestick data from the given dataset

void computation\_weather\_data::compute\_candlestick\_yearly(const std::vector<std::string>& dataset) {

    // User input of the country code

    std::cout << "Enter the country code: ";

    std::string Code;

    std::cin >> Code;

    // Add \_temperature with the country code to find the column

    std::string col\_name = Code + "\_temperature";

    // Find the index of the column

    int col\_Index = CSVReader::Col\_Index(col\_name);

    // If the country code is not found, print the message

    if (col\_Index == -1)

    {

        std::cout << "Wrong Input! Country code is not from the list stated above" << std::endl;

        return;

    }

    // Utils functions to group data and compute candlesticks.

    auto Temps\_yearly = CSVReader::group\_temp\_year(dataset, col\_Index);

    auto candlesticks = CSVReader::computeCandlesticks(Temps\_yearly);

    // Print the values of the candlestick

    std::cout << "======================================================================" << std::endl;

    std::cout << "                    Candlestick data for " << Code << std::endl;

    std::cout << "======================================================================" << std::endl;

    for (const auto &candle : candlesticks)

    {

        std::cout << "Date: " << candle.date << "| Open: " << candle.open << "| High: " << candle.high << "| Low: " << candle.low << "| Close: " << candle.close << std::endl;

    }

};

**// TASK 2**

void computation\_weather\_data::text\_plot\_yearly(const std::vector<std::string>& dataset) {

    // User input multiple country codes

    std::cout << "Enter country codes (comma-separated): ";

    std::string input;

    std::cin.ignore();

    std::getline(std::cin, input);

    // Use the helper function

    auto code\_to\_index = CSVReader::CountryCode\_processing(input);

    // Process each country code

    for (const auto& pair : code\_to\_index) {

        const std::string& countryCode = pair.first;

        int col\_index = pair.second;

        // Utils functions to group data and compute candlesticks.

        auto Temps\_yearly = CSVReader::group\_temp\_year(dataset, col\_index);

        auto candlesticks = CSVReader::computeCandlesticks(Temps\_yearly);

        Candlestick::yearly\_graph(candlesticks, countryCode);

    }

}

**// TASK 3**

void computation\_weather\_data::text\_plot\_Filters(const std::vector<std::string>& dataset) {

    // User input for country codes

    std::cout << "Enter country codes (comma-separated): ";

    std::string input;

    std::cin.ignore();

    std::getline(std::cin, input);

    // User input for year range

    std::string year\_Range;

    std::cout << "Enter the year range (e.g., 1980-1983): ";

    std::cin >> year\_Range;

    // Parse the year range

    int dash = year\_Range.find('-');

    if (dash == std::string::npos) {

        std::cout << "Error: Invalid year range format!" << std::endl;

        return;

    }

    int startYear = std::stoi(year\_Range.substr(0, dash));

    int endYear = std::stoi(year\_Range.substr(dash + 1));

    // Error message for invalid year range

    if (startYear > endYear) {

        std::cout << "Error: Start year cannot be greater than end year!" << std::endl;

        return;

    }

    // Process country codes and find their column indices

    std::map<std::string, int> code\_to\_index = CSVReader::CountryCode\_processing(input);

    // Iterate over processed country codes and their column indices

    for (const auto& pair : code\_to\_index) {

        const std::string& countryCode = pair.first;

        int col\_index = pair.second;

        // Group temperatures by year within the range

        auto Temps\_yearly = CSVReader::group\_temp\_year\_range(dataset, col\_index, startYear, endYear);

        // Compute candlesticks from grouped temperatures

        auto candlesticks = CSVReader::computeCandlesticks(Temps\_yearly);

        Candlestick::Filter\_graph(candlesticks, countryCode);

     }

}

**// TASK 4**

void computation\_weather\_data::Future\_Prediction(const std::vector<std::string>& dataset) {

    // User input of country code for the future prediction

    std::cout << "Enter the country code for future temperature prediction: ";

    std::string countryCode;

    std::cin >> countryCode;

    // Add the country code with \_temperature and look for it in the index of the dataset

    std::string col\_name = countryCode + "\_temperature";

    int col\_index = CSVReader::Col\_Index(col\_name);

    // Wrong input

    if (col\_index == -1) {

        std::cout << "Error: Wrong Country Code" << std::endl;

        return;

    }

    // User input the number of years to predict

    std::cout << "Enter the number of years to predict (e.g., 6 for 2020 to 2025): ";

    int years\_predicted;

    std::cin >> years\_predicted;

    // If the user input negative number error message is printed

    if (years\_predicted <= 0) {

        std::cout << "Error: The number of years to predict must be greater than 0." << std::endl;

        return;

    }

    // Function to group temperature data by year

    auto Temps\_yearly = CSVReader::group\_temp\_year(dataset, col\_index);

    // Store year statistics and list of years

    std::vector<int> years;

    std::map<int, double> avg\_temps;

    std::map<int, double> yearlyHighs;

    std::map<int, double> yearlyLows;

    // Calculate average, high, low temperatures for each year

    for (const auto& entry : Temps\_yearly) {

        double averageTemp = std::accumulate(entry.second.begin(), entry.second.end(), 0.0) / entry.second.size();

        double yearlyHigh = \*std::max\_element(entry.second.begin(), entry.second.end());

        double yearlyLow = \*std::min\_element(entry.second.begin(), entry.second.end());

        avg\_temps[entry.first] = averageTemp;

        yearlyHighs[entry.first] = yearlyHigh;

        yearlyLows[entry.first] = yearlyLow;

        years.push\_back(entry.first);

    }

    // Sort the years in ascending order

    std::sort(years.begin(), years.end());

    // Linear regression to predict future temperatures

    double sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;

    int n = years.size();

    for (int i = 0; i < n; i++) {

        int x = years[i];

        // Average temperature for the year

        double y = avg\_temps[years[i]];

        sumX += x;

        sumY += y;

        sumXY += x \* y;

        sumX2 += x \* x;

    }

    // Calculate the slope

    double m = (n \* sumXY - sumX \* sumY) / (n \* sumX2 - sumX \* sumX);

    // Values for candlestick between year 2015 to 2019, the last 5 years

    std::vector<Candlestick> prev\_candlestickdata;

    double totalHighDiff = 0, totalLowDiff = 0;

    int count = 0;

    // Generate candlesticks for 2015 to 2019

    for (int i = 0; i < years.size(); i++) {

        if (years[i] >= 2015 && years[i] <= 2019) {

            // Temperature data for the year

            auto& temps = Temps\_yearly[years[i]];

            double prev\_close = (i == 0) ? 0.0 : avg\_temps[years[i - 1]];

            Candlestick candle = Candlestick::compute(std::to\_string(years[i]), temps, prev\_close);

            // Difference in highs

            totalHighDiff += (yearlyHighs[years[i]] - yearlyHighs[years[i - 1]]);

            // Difference in low

            totalLowDiff += (yearlyLows[years[i]] - yearlyLows[years[i - 1]]);

            count++;

            prev\_candlestickdata.push\_back(candle);

        }

    }

    // Average difference in highs and lows

    double avgHighDiff = totalHighDiff / count;

    double avgLowDiff = totalLowDiff / count;

    std::vector<Candlestick> future\_Candlesticks;

    double prev\_close = prev\_candlestickdata.back().close;

    // Generate predicted candlesticks

    for (int i = 1; i <= years\_predicted; i++) {

        int futureYear = 2019 + i;

        // Predict average temperature

        double Temp\_predicted = avg\_temps[2019] + m \* i;

        double open = prev\_close;

        double close = Temp\_predicted;

        // Predict high and low temperature

        double high = yearlyHighs[2019] + avgHighDiff \* i;

        double low = yearlyLows[2019] + avgLowDiff \* i;

        Candlestick futureCandle(std::to\_string(futureYear), open, high, low, close);

        future\_Candlesticks.push\_back(futureCandle);

        prev\_close = close;

    }

    // Combine past and future candlesticks for the chart

    std::vector<Candlestick> combined\_Candlesticks = prev\_candlestickdata;

    combined\_Candlesticks.insert(combined\_Candlesticks.end(), future\_Candlesticks.begin(), future\_Candlesticks.end());

    // Display values for past and future candlesticks

    for (const auto& candle : combined\_Candlesticks) {

        std::cout << "Year: " << candle.date << " | Open: " << candle.open << " | High: " << candle.high << " | Low: " << candle.low << " | Close: " << candle.close << std::endl;

    }

    // Future graph

    Candlestick::future\_graph(combined\_Candlesticks, countryCode);

}

**merkelmain.h**

// Include all the libraries required to run the program

#pragma once

#include "Candlestick.h"

class MerkelMain {

public:

    MerkelMain();

    void init();

    // Static functions

    static std::string header\_Row;

    static std::vector<std::string> dataset;

    static std::vector<std::string> available\_Codes;

    // Computation functions

    void compute\_candlestick\_yearly();

    void text\_plot\_yearly();

    void text\_plot\_Filters();

    void Future\_Prediction();

private:

    // Main menu

**// Followed starter code**

    void printMenu();

    // Process the user option

    void processUserOption(int userOption);

**// End following starter code**

    // Display the country codes

    void show\_codes();

**// Followed starter code**

    // Show the user how to use the application

    void Menuinstruction();

**// End following starter code**

    // Exit option

    void Exit();

};

**merkelmain.cpp**

// Include all the libraries and the header file required

#include <iostream>

#include "CSVReader.h"

#include "MerkelMain.h"

#include "Weatherdata.h"

// Static members of MerkelMain class

std::string MerkelMain::header\_Row;

std::vector<std::string> MerkelMain::dataset;

std::vector<std::string> MerkelMain::available\_Codes;

**// Followed starter code**

// Constructor for Merkelmain

MerkelMain::MerkelMain() {}

// Init function of the program

void MerkelMain::init()

{

    // Stores user input

    int input;

    while (true)

    {

        // Show the menu options

        printMenu();

        // Get the user input

        std::cin >> input;

        // Process the selected option form the user

        processUserOption(input);

    }

}

// Menu function - Prints the different menu options

void MerkelMain::printMenu()

{

    std::cout << "===================================" << std::endl;

    std::cout << "               Menu"                 << std::endl;

    std::cout << "===================================" << std::endl;

    std::cout << "1: View Available Country Code" << std::endl;

    std::cout << "2: Compute candlestick data Yearly" << std::endl;

    std::cout << "3: Text Based Plot Yearly" << std::endl;

    std::cout << "4: Text Based Plot Using Filters" << std::endl;

    std::cout << "5: Future Prediction" << std::endl;

    std::cout << "6: Help Center" << std::endl;

    std::cout << "7: Exit" << std::endl;

    std::cout << "===================================" << std::endl;

}

**// End following starter code**

// Show the available country codes for the inputs

void MerkelMain::show\_codes()

{

    // Loading the dataset

    dataset = CSVReader::readCSV("weather\_data\_EU\_1980-2019\_temp\_only.csv");

    // Temporary vector to store column headers

    std::vector<std::string> col\_header;

    // Temporary string to parse headers

    std::string temporary;

    // Split the headerrow into individual column headers

    for (size\_t i = 0; i < header\_Row.length(); ++i)

    {

        if (header\_Row[i] == ',')

        {

            col\_header.push\_back(temporary);

            temporary.clear();

        }

        else

        {

            temporary += header\_Row[i];

        }

    }

    if (!temporary.empty())

    {

        col\_header.push\_back(temporary);

    }

    available\_Codes.clear();

    // Extract country codes from headers with the temperature at the bacm

    for (size\_t i = 1; i < col\_header.size(); ++i)

    {

        if (col\_header[i].find("\_temperature") != std::string::npos)

        {

            available\_Codes.push\_back(col\_header[i].substr(0, col\_header[i].find("\_temperature")));

        }

    }

    // Print the country codes

    std::cout << "Available Country Codes: ";

    for (size\_t i = 0; i < available\_Codes.size(); ++i) {

        std::cout << available\_Codes[i];

        if (i != available\_Codes.size() - 1) {

            std::cout << ",";

        }

    }

    std::cout << std::endl;

}

// Function to compute yearly candlestick data

void MerkelMain::compute\_candlestick\_yearly() {

    // Print the country codes first to see what codes are available for processing

    show\_codes();

    // Non-static function, so created a variable called weather processor to access the function.

    computation\_weather\_data computation;

    computation.compute\_candlestick\_yearly(dataset);

}

// Function to generate a yearly text based plot

void MerkelMain::text\_plot\_yearly() {

    // Print the country codes first to see what codes are available for processing

    show\_codes();

    // Non-static function, so created a variable called weather processor to access the function.

    computation\_weather\_data computation;

    computation.text\_plot\_yearly(dataset);

}

// Function to generate a filtered text based plot

void MerkelMain::text\_plot\_Filters() {

    // Print the country codes first to see what codes are available for processing

    show\_codes();

    // Non-static function, so created a variable called weather processor to access the function.

    computation\_weather\_data computation;

    computation.text\_plot\_Filters(dataset);

}

// Function to predict the future temperature and plot in a text based plot

void MerkelMain::Future\_Prediction() {

    // Print the country codes first to see what codes are available for processing

    show\_codes();

    // Non-static function, so created a variable called weather processor to access the function.

    computation\_weather\_data computation;

    computation.Future\_Prediction(dataset);

}

// Function to instruct the user on how to use the menu options

void MerkelMain::Menuinstruction()

{

    std::cout << "===================================" << std::endl;

    std::cout << "             Instructions          " << std::endl;

    std::cout << "===================================" << std::endl;

    std::cout << "Welcome! This program leads you to explore and filter temperature data from various countries over different time periods." << std::endl;

    std::cout << std::endl;

    std::cout << "Overview:" << std::endl;

    std::cout << "1. View computations for all country codes across all years from 1980 to 2019. " << std::endl;

    std::cout << "2. Filter Data: Refine the dataset with various filter options:" << std::endl;

    std::cout << "   - By Country: Focus on temperature records for a specific country." << std::endl;

    std::cout << "   - By Year: Retrieve data for a specific year range." << std::endl;

    std::cout << "3. Predict Future Trends: Use historical temperature patterns to forecast future temperatures for selected countries." << std::endl;

    std::cout << std::endl;

    std::cout << "How to Use:" <<std::endl;

    std::cout << "   - Select options from the main menu by entering the menu options" << std::endl;

    std::cout << "   - Follow the instructions to filter or analyze the data as required." << std::endl;

    std::cout << std::endl;

    std::cout << "For the sample graph, please open the sample\_graph.txt file to view the details of the graph." << std::endl;

    std::cout << std::endl;

    std::cout << "Thank you for using the program!" << std::endl;

}

// Function to exit the program

void MerkelMain::Exit()

{

    std::cout << "Thank you. Exiting now..." << std::endl;

    exit(0);

}

// Function to process the selected menu option

**// Followed starter code**

void MerkelMain::processUserOption(int userOption)

{

    if (userOption == 0 || userOption > 7)

    {

        std::cout << "Invalid choice. Choose 1-7" << std::endl;

    }

    if (userOption == 1)

    {

        show\_codes();

    }

    if (userOption == 2)

    {

        compute\_candlestick\_yearly();

    }

    if (userOption == 3)

    {

        text\_plot\_yearly();

    }

    if (userOption == 4)

    {

        text\_plot\_Filters();

    }

    if (userOption == 5)

    {

        Future\_Prediction();

    }

    if (userOption == 6)

    {

        Menuinstruction();

    }

    if (userOption == 7)

    {

        Exit();

    }

}

**// End following starter code**