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