Introduction

The main objective of this project is to use text-based plotting and predictive analysis to examine historical weather data, extract insightful information, and visualize trends. This report outlines the design and implementation of a C++ program that processes weather data stored in CSV format. The program performs several key tasks, including computing candlestick data, filtering and plotting temperature trends, and predicting future average temperatures using historical data.

The project highlights the application of key programming concepts:

1. File Handling: Efficiently reading and parsing CSV files to extract and process data.
2. Data Structures: Utilizing vectors, maps, and structs for organized and efficient data management.
3. Algorithm Development: Implementing candlestick calculations and polynomial regression for predictive analysis.
4. Text-Based Visualization: Presenting temperature trends in an intuitive, text-based format.
5. Error Handling: Managing missing or invalid data and ensuring robust user input validation.

The tasks in the program are organised to guarantee code clarity and reusability. The program allows users to:

1. Compute candlestick summaries of temperature trends.
2. Plot temperature data grouped by decades in a text-based format.
3. Filter and visualize data based on user-defined parameters (e.g., country, date range, temperature).
4. Predict future temperature trends using polynomial regression.

This report provides an elaborate explanation of each task's implementation, supported by code snippets, results, and visualizations. It also discusses the difficulties encountered during development and the solutions that were used.

Task 1: Compute Candlestick Data

Description

The program processes historical weather data from a CSV file to compute candlestick data for a specific country and to summarize yearly temperature trends. Each candlestick includes:

1. Open: The first recorded temperature of the year.
2. Close: The last recorded temperature of the year.
3. High: The highest recorded temperature of the year.
4. Low: The lowest recorded temperature of the year.

Implementation

The computeCandlestickData function processes the CSV data by:

1. Grouping temperatures by year.
2. Calculating metrics (open, high, low, close) for each year.
3. Returning the computed data as a vector of Candlestick structs.

The function supports filtering by country through a prefix (e.g., "AT" for Austria) to ensure efficient data processing. The use of dynamic column indexing enables accurate extraction of temperature data based on country-specific columns.

Below are the relevant code snippets for this task.

File: “main.cpp”

```
// --- Task 1: Candlestick Data Computation ---

    try {
        // Compute candlestick data for Austria ("AT") grouped by year
        std::cout << "\nComputing candlestick data for Austria (AT) by
year...\n";
        auto candlesticks = computeCandlestickData(data, "AT", "year");

        if (candlesticks.empty()) {
            std::cerr << "No candlestick data could be computed. Check input
data.\n";
            return 1;
        }

        // Display the computed candlestick data
        std::cout << "\nComputed Candlestick Data:\n";
        for (const auto& candle : candlesticks) {
            std::cout << "Date: " << candle.date
                      << ", Open: " << candle.open
                      << ", High: " << candle.high
                      << ", Low: " << candle.low
                      << ", Close: " << candle.close << std::endl;
        }
```

This snippet demonstrates how candlestick data is computed for Austria and displayed to the user. It calls the computeCandlestickData function with parameters ("AT", "year") to process the dataset for Austria and group data by year. The function computeCandlestickData is defined in utils.cpp and prototyped in utils.h.

File: “utils.cpp”

```
// --- Task 1: Candlestick Data Computation ---

/**
 * Computes candlestick data for a specific country and time frame.
 *
 * @param data The dataset as a 2D vector of strings.
 * @param country_prefix The prefix for the country (e.g., "AT" for
Austria).
 * @param time_frame The time frame for aggregation ("year", "month", or
"day").
 * @return A vector of computed candlestick objects.
 */
std::vector<Candlestick> computeCandlestickData(
    const std::vector<std::vector<std::string>> &data,
    const std::string &country_prefix,
    const std::string &time_frame) {
    std::map<std::string, std::vector<double>> grouped_data;
    int temp_column = -1;

    // Identify the temperature column
    for (size_t i = 0; i < data[0].size(); ++i) {
        if (data[0][i] == country_prefix + "_temperature") {
            temp_column = i;
            break;
        }
    }

    if (temp_column == -1) {
        throw std::runtime_error("Temperature column not found for " +
country_prefix);
    }

    // Group data based on the specified time frame
    for (size_t i = 1; i < data.size(); ++i) {
        std::string timestamp = data[i][0];
        std::string time_key;

        if (time_frame == "year") {
            time_key = timestamp.substr(0, 4);
        } else if (time_frame == "month") {
            time_key = timestamp.substr(0, 7);
        } else if (time_frame == "day") {
            time_key = timestamp.substr(0, 10);
        }

        try {
            double temp = std::stod(data[i][temp_column]);
            grouped_data[time_key].push_back(temp);
```

```
        } catch (const std::exception &) {
            std::cerr << "Invalid temperature data: Skipping row " << i <<
std::endl;
        }
    }

    // Compute candlestick metrics for each group
    std::vector<Candlestick> candlesticks;
    for (const auto &[key, temps] : grouped_data) {
        double open = temps.front();
        double close = temps.back();
        double high = *std::max_element(temps.begin(), temps.end());
        double low = *std::min_element(temps.begin(), temps.end());
        candlesticks.push_back({key, open, high, low, close});
    }

    return candlesticks;
}
```

This snippet identifies the column for the selected country's temperature data in the CSV file. It iterates through the header row of the CSV file to find the column matching country_prefix + "_temperature". It is called in main.cpp during candlestick computation.

File “utils.h”:

```
// --- Task 1: Candlestick Data Computation ---

/**
 * Computes candlestick data for a given country and time frame.
 *
 * @param data The dataset as a 2D vector of strings.
 * @param country_prefix The country prefix (e.g., "AT" for Austria).
 * @param time_frame The time frame (e.g., "year", "month", or "day").
 * @return A vector of computed Candlestick objects.
 */
std::vector<Candlestick> computeCandlestickData(
    const std::vector<std::vector<std::string>> &data,
    const std::string &country_prefix,
    const std::string &time_frame
);
```

This snippet groups candlestick data by decade and visualizes each group using text-based plots. It calls groupByDecade to cluster data into decades. It iterates through each group, extracting the decade year from the first entry and plotting it using plotCandlestickGroup. It uses groupByDecade and plotCandlestickGroup for grouping and visualization.

File “Candlestick.h”:

```
// --- Task 1: Candlestick Data Computation ---

struct Candlestick {
    std::string date;  // Time frame (e.g., year, month, or day).
    double open;       // Opening temperature.
    double high;       // Highest temperature.
    double low;        // Lowest temperature.
    double close;      // Closing temperature.

    /**
     * @brief Constructs a Candlestick instance.
     *
     * @param d The time frame (e.g., year, month, or day).
     * @param o The opening temperature.
     * @param h The highest temperature.
     * @param l The lowest temperature.
     * @param c The closing temperature.
     */
    Candlestick(std::string d, double o, double h, double l, double c)
        : date(d), open(o), high(h), low(l), close(c) {}
};
```

In this snippet, the Candlestick struct models summarized temperature data for a specific time frame (e.g., year, month, or day). It includes key metrics—open, high, low, and close—to represent the first, highest, lowest, and last recorded temperatures, respectively. The constructor initializes these fields, ensuring structured and consistent data for further computation and visualization.

Output for Task 1:

```
Computing candlestick data for Austria (AT) by year...

Computed Candlestick Data:
Date: 1980, Open: -3.64, High: 29.132, Low: -14.507, Close: -0.989
Date: 1981, Open: -1.145, High: 29.419, Low: -16.219, Close: 0.822
Date: 1982, Open: 0.367, High: 28.395, Low: -15.084, Close: -5.923
Date: 1983, Open: -5.873, High: 32.416, Low: -18.098, Close: 0.057
Date: 1984, Open: 0.212, High: 32.659, Low: -13.338, Close: -10.377
Date: 1985, Open: -10.106, High: 29.166, Low: -22.705, Close: -8.16
Date: 1986, Open: -8.188, High: 29.785, Low: -20.601, Close: 1.875
Date: 1987, Open: 1.904, High: 28.054, Low: -25.301, Close: 0.874
Date: 1988, Open: 0.793, High: 31.313, Low: -13.019, Close: -1.837
Date: 1989, Open: -2.226, High: 28.968, Low: -10.969, Close: -5.765
Date: 1990, Open: -5.823, High: 29.145, Low: -11.347, Close: 0.41
Date: 1991, Open: 0.413, High: 30.448, Low: -15.978, Close: -7.615
Date: 1992, Open: -7.68, High: 32.326, Low: -13.679, Close: -10.272
Date: 1993, Open: -10.389, High: 30.092, Low: -17.096, Close: -0.568
Date: 1994, Open: -0.487, High: 31.458, Low: -12.935, Close: -1.311
Date: 1995, Open: -1.088, High: 31.62, Low: -14.468, Close: -3.946
Date: 1996, Open: -4.04, High: 27.225, Low: -20.283, Close: -12.245
Date: 1997, Open: -12.095, High: 27.528, Low: -12.095, Close: -1.116
Date: 1998, Open: -1.135, High: 31.945, Low: -15.665, Close: -3.564
Date: 1999, Open: -3.534, High: 29.667, Low: -16.172, Close: -8.687
Date: 2000, Open: -8.604, High: 32.539, Low: -16.425, Close: -10.101
```

```
Date: 2000, Open: -8.604, High: 32.539, Low: -16.425, Close: -10.101
Date: 2001, Open: -10.671, High: 30.489, Low: -17.771, Close: -8.538
Date: 2002, Open: -9.329, High: 30.305, Low: -16.596, Close: -5.478
Date: 2003, Open: -5.623, High: 33.745, Low: -17.103, Close: -3.556
Date: 2004, Open: -3.767, High: 28.997, Low: -15.015, Close: -1.328
Date: 2005, Open: -1.25, High: 31.314, Low: -19.122, Close: -4.165
Date: 2006, Open: -4.404, High: 30.55, Low: -21.521, Close: 2.016
Date: 2007, Open: 1.95, High: 32.701, Low: -10.583, Close: -5.201
Date: 2008, Open: -5.437, High: 28.851, Low: -11.306, Close: -7.521
Date: 2009, Open: -7.518, High: 29.917, Low: -15.502, Close: 0.522
Date: 2010, Open: 0.412, High: 30.419, Low: -18.188, Close: -5.356
Date: 2011, Open: -5.611, High: 32.679, Low: -13.038, Close: -0.975
Date: 2012, Open: -0.816, High: 32.54, Low: -17.832, Close: -2.855
Date: 2013, Open: -3.024, High: 32.556, Low: -15.969, Close: -2.293
Date: 2014, Open: -2.411, High: 29.589, Low: -9.64, Close: -6.495
Date: 2015, Open: -6.503, High: 32.577, Low: -11.174, Close: -2.239
Date: 2016, Open: -2.145, High: 29.467, Low: -12.179, Close: -4.59
Date: 2017, Open: -4.64, High: 32.631, Low: -14.708, Close: -0.289
Date: 2018, Open: -0.508, High: 30.307, Low: -17.596, Close: -1.666
Date: 2019, Open: -1.524, High: 31.839, Low: -10.777, Close: -2.271
```

The output displays candlestick data for Austria (AT) by year from 1980 to 2019, summarizing yearly temperature trends with four metrics: Open, High, Low, and Close.

Challenges and Solutions

1. Challenge 1: Handling missing or invalid temperature data in the dataset.

This was addressed by implementing robust error handling to skip rows with non-numeric values, ensuring that only valid entries are processed.

2. Challenge 2: Identifying the correct column for a country's temperature data in the CSV file.

This was resolved by employing dynamic column indexing, which accurately identifies the relevant data columns for processing by scanning the header row for the country-specific prefix.

Key Insights

1. Clarity: Candlestick data provides an intuitive and concise summary of yearly temperature trends, making it easier to analyze large datasets.

2. Grouping Benefits: Aggregating data by year allows for meaningful year-over-year trend analysis, highlighting long-term patterns and anomalies.

Task 2: Create a Text-Based Plot of the Candlestick Data

Description

This task visualizes candlestick data in a text-based format, providing an accessible method to analyze temperature trends over decades. Each candlestick is represented vertically using symbols:

- **High (*):** Represents the highest temperature recorded in a year.
- **Low (*):** Represents the lowest temperature recorded in a year.
- **Open (O):** The first recorded temperature of the year.
- **Close (C):** The last recorded temperature of the year.
- **|:** Represents the range between high and low temperatures.

To simplify long-term trend analysis, candlesticks are grouped by decade for easy visualization and comparison.

Implementation

The plotGroupedCandlesticks function groups the computed candlesticks by decade using the helper function groupByDecade. It then calls plotCandlestickGroup to generate a dynamic text-based plot for each decade. The plot automatically adjusts its height based on the range of temperatures in the dataset, ensuring clear and proportional visualization of trends.

This design preserves readability and simplicity while enabling users to see patterns and variances across a number of years.

Below are the code snippets for the plotting logic.

File: “main.cpp”

```
// --- Task 2: Plot Candlestick Data ---

        // Plot the candlestick data grouped by decade
        std::cout << "\nText-Based Plot of Candlesticks for Austria (AT) by
Decade:\n";
        std::cout << "-----------------------------------\n";
        plotGroupedCandlesticks(candlesticks);

        // Main Menu for User Actions
        char proceed;
        do {
            std::cout << "\nChoose an option:\n";
            std::cout << "1. Filter and plot data (Task 3)\n";
            std::cout << "2. Predict temperatures (Task 4)\n";
            std::cout << "0. Exit\n";
            std::cout << "Enter your choice: ";
            int choice;
            std::cin >> choice;
```

This snippet initiates the plotting process by calling the plotGroupedCandlesticks function with the computed candlestick data as input. The function groups the candlestick data by decade and visualizes each group with a text-based plot.

File: “utils.cpp”

```
// --- Task 2: Plotting Functions ---

/**
 * Groups candlesticks by decade.
 *
 * @param candlesticks A vector of candlestick data to be grouped.
 * @return A vector of grouped candlesticks, where each inner vector
contains data for a single decade.
 */
std::vector<std::vector<Candlestick>> groupByDecade(const
std::vector<Candlestick>& candlesticks) {
    std::vector<std::vector<Candlestick>> grouped;
    int current_decade = -1;
    std::vector<Candlestick> current_group;

    for (const auto& candle : candlesticks) {
        int year = std::stoi(candle.date); // Convert date to integer year
        int decade = (year / 10) * 10;

        if (decade != current_decade) {
            if (!current_group.empty()) {
                grouped.push_back(current_group);
```

```
            }
            current_group.clear();
            current_decade = decade;
        }

        current_group.push_back(candle);
    }

    if (!current_group.empty()) {
        grouped.push_back(current_group);
    }

    return grouped;
}

/**
 * Plots a single group of candlesticks with a text-based visualization.
 *
 * @param candlesticks A vector of candlestick data to plot.
 * @param plot_height The height of the plot (number of rows in the output).
 */
void plotCandlestickGroup(const std::vector<Candlestick>& candlesticks, int
plot_height) {
    if (candlesticks.empty()) {
        std::cout << "No candlestick data to plot.\n";
        return;
    }

    double global_high = -1e9, global_low = 1e9;

    // Find the high and low across Europe for the filtered candlesticks
    for (const auto& candle : candlesticks) {
        global_high = std::max(global_high, candle.high);
        global_low = std::min(global_low, candle.low);
    }

    double range = global_high - global_low;
    if (range == 0) range = 1; // Prevent division by zero

    // Adjust plot height dynamically based on range
    int adjusted_plot_height = std::min(plot_height, static_cast<int>(range
* 2));
    adjusted_plot_height = std::max(adjusted_plot_height, 10); // Ensure a
minimum plot height

    // Print y-axis labels and candlesticks
    for (int i = adjusted_plot_height; i >= 0; --i) {
        double temp = global_low + (i * range / adjusted_plot_height);
        std::cout << std::setw(5) << std::fixed << std::setprecision(1) <<
temp << " | ";
```

```
        for (const auto& candle : candlesticks) {
            double open_pos = (candle.open - global_low) / range *
adjusted_plot_height;
            double close_pos = (candle.close - global_low) / range *
adjusted_plot_height;
            double high_pos = (candle.high - global_low) / range *
adjusted_plot_height;
            double low_pos = (candle.low - global_low) / range *
adjusted_plot_height;

            if (i == static_cast<int>(high_pos)) {
                std::cout << "*      "; // High
            } else if (i == static_cast<int>(low_pos)) {
                std::cout << "*      "; // Low
            } else if (i == static_cast<int>(open_pos)) {
                std::cout << "O      "; // Open
            } else if (i == static_cast<int>(close_pos)) {
                std::cout << "C      "; // Close
            } else if (i < static_cast<int>(high_pos) && i >
static_cast<int>(low_pos)) {
                std::cout << "|      "; // Vertical stalk
            } else {
                std::cout << "       "; // Empty space
            }
        }
        std::cout << "\n";
    }

    // Print year labels below the plot
    std::cout << "     "; // Space for the y-axis labels
    for (size_t i = 0; i < candlesticks.size(); ++i) {
        std::cout << std::setw(7) << candlesticks[i].date;
    }
    std::cout << "\n";
}

/**
 * Plots grouped candlesticks by decade with a text-based visualization.
 *
 * @param candlesticks A vector of candlestick data to group and plot.
 * @param plot_height The height of the plot for each group (number of rows
in the output).
 */
void plotGroupedCandlesticks(const std::vector<Candlestick>& candlesticks,
int plot_height) {
    auto grouped = groupByDecade(candlesticks);

    for (const auto& group : grouped) {
        if (!group.empty()) {
```

```
            int decade = std::stoi(group[0].date) / 10 * 10;
            std::cout << "\nCandlestick Data for " << decade << "s:\n";
            plotCandlestickGroup(group, plot_height);
            std::cout << "-----------------------------------\n";
        }
    }

}
```

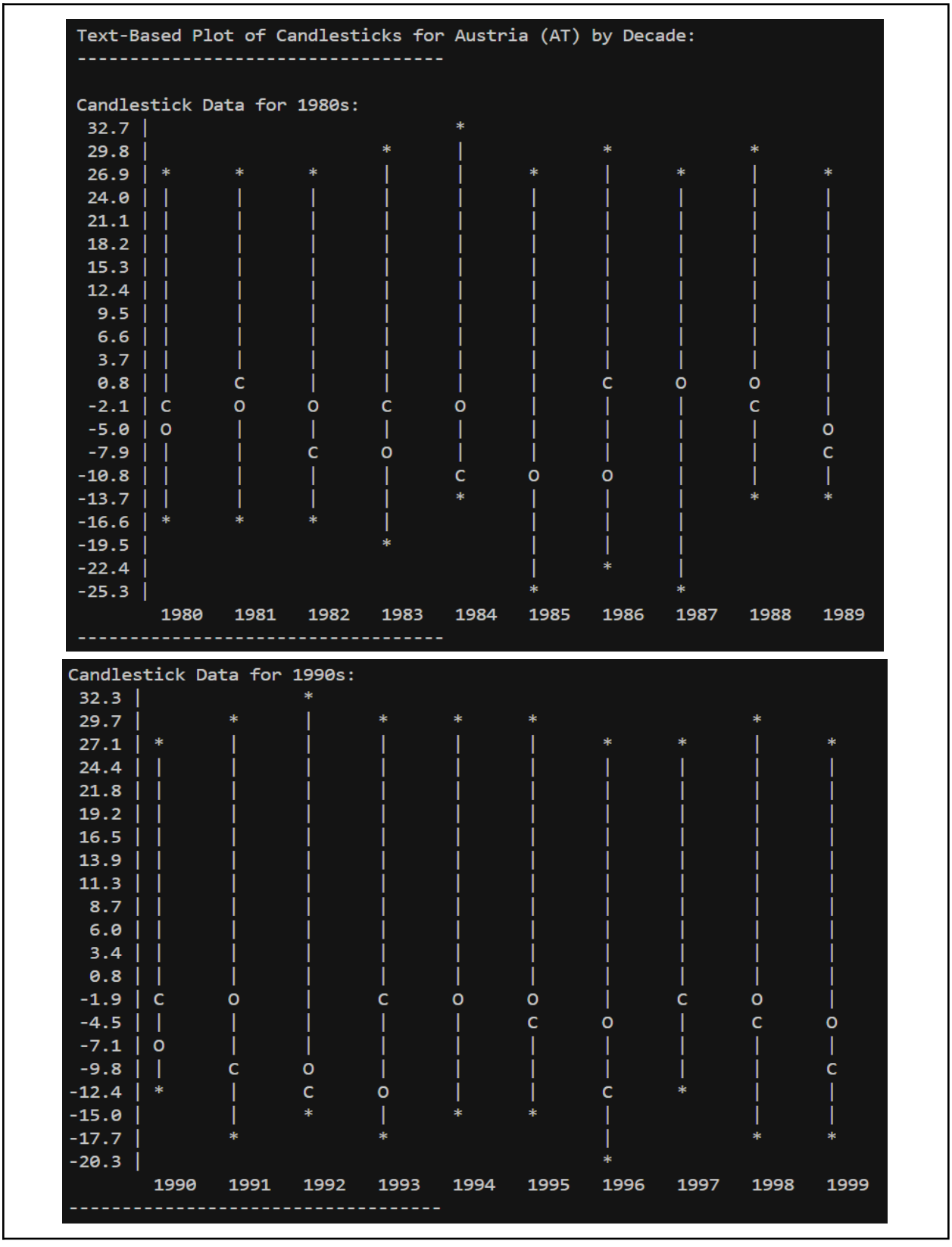
This snippet groups candlestick data by decade to simplify long-term trend analysis. It iterates through the candlestick data, calculates the decade for each year and groups candlesticks with the same decade. It then creates a dynamic text-based plot for a single group of candlesticks, representing metrics like high (*), low (*), open (O), and close (C). The plotGroupedCandlesticks function then combines the above functions to plot candlesticks grouped by decade with clear separation for each group.

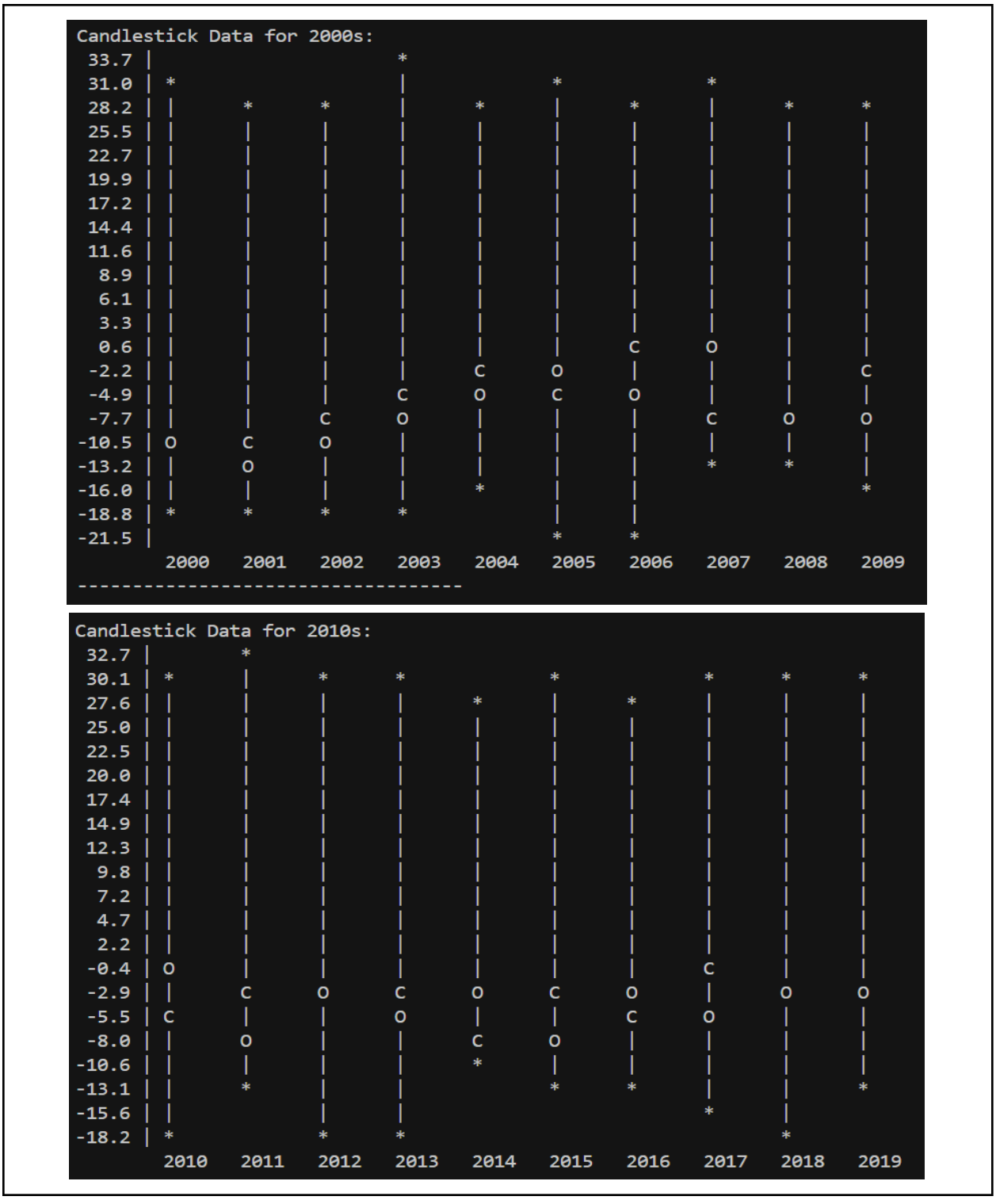
File “utils.h”:

```
// --- Task 2: Plotting Functions ---
/**
 * Plots candlestick data as a text-based graph.
 *
 * @param candlesticks A vector of Candlestick objects to plot.
 */
void plotCandlesticks(const std::vector<Candlestick>& candlesticks);

/**
 * Creates a grouped text-based plot of candlestick data.
 *
 * @param candlesticks A vector of Candlestick objects to plot.
 * @param plot_height The height of the plot.
 */
void plotGroupedCandlesticks(const std::vector<Candlestick>& candlesticks,
int plot_height = 20);
```

This snippet declares the plotting-related functions (plotCandlesticks and plotGroupedCandlesticks) for use in the program, ensuring modularity and reusability.

Output for Task 2:





The output is a text-based plot of candlestick data grouped by decade, with a dynamically scaled y-axis showing temperature ranges. Each year is represented vertically using symbols: * for high/low, O for the first recorded temperature, and C for the last. The x-axis lists the years within the decade, aligning each year with its candlestick. This visualization effectively highlights temperature trends, fluctuations, and patterns over decades.

Challenges and Solutions

1. Challenge 1: Scaling the y-axis dynamically for varying temperature ranges.

This was addressed by calculating the global high and low temperatures for each group and dynamically scaling the y-axis to fit the range, ensuring proportional and readable plots.

2. Challenge 2: Handling empty or invalid candlestick data.

This was addressed by implementing checks to detect and handle cases where no valid data is available, displaying appropriate error messages to inform users.

Key Insights

1. Visualization Simplicity: Text-based plots offer an effective and lightweight alternative for visualizing data without requiring external libraries.

2. Trend Analysis: Grouping candlestick data by decades makes it easier to observe long-term patterns and temperature fluctuations.

Task 3: Filtering Options and Plotting a Text-Based Graph

Description

The third task focuses on enhancing program usability by allowing users to filter candlestick data based on specific criteria:

1. Filter by Country: Filters data for a selected country using its prefix.
2. Filter by Date Range: Allows the user to specify a start and end date for filtering.
3. Filter by Temperature Range: Filters candlestick data for a specified minimum and maximum temperature range.

After filtering, a text-based plot of the filtered candlestick data is generated, providing a visual representation similar to Task 2.

Implementation

To implement these filtering options, the program includes the following main functions:

1. filterByCountry: Extracts data for a specified country using a prefix (e.g., "AT" for Austria).
2. filterByDateRange: Filters data between a user-defined start and end date.
3. filterByTemperatureRange: Filters candlestick data to include only entries within a specific temperature range.

These functions are integrated with user input prompts in the main.cpp file, allowing dynamic data selection and filtering.

These filters are implemented as follows.

File: “main.cpp”

```
// --- Task 3: Filtering Options ---

            switch (choice) {
                case 1: {
                    std::cout << "\nWould you like to filter the data?
(y/n): ";
                    char filter_choice;
                    std::cin >> filter_choice;

                    if (filter_choice == 'y' || filter_choice == 'Y') {
                        std::cout << "\nChoose a filtering option:\n";
                        std::cout << "1. Filter by country\n";
                        std::cout << "2. Filter by date range\n";
                        std::cout << "3. Filter by temperature range\n";
                        std::cout << "Enter your choice: ";
                        int filter_option;
                        std::cin >> filter_option;

                        std::vector<Candlestick> filtered_data;

                        switch (filter_option) {
                            case 1: {
                                // Display available countries
                                displayAvailableCountries(data);

                                // Filter by country
                                std::string country_prefix;
                                std::cout << "(Kindly input in
UPPERCASE)\n";
                                std::cout << "Enter the country prefix
(e.g., 'AT' for Austria):";
                                std::cin >> country_prefix;
                                filtered_data = filterByCountry(data,
country_prefix, "year");
                                break;
                            }
                            case 2: {
                                // Display available date range
                                displayAvailableDateRange(data);

                                // Filter by date range
                                std::string start_date, end_date;
                                std::cout << "Enter start date (YYYY): ";
                                std::cin >> start_date;
                                std::cout << "Enter end date (YYYY): ";
                                std::cin >> end_date;
                                filtered_data =
filterByDateRange(candlesticks, start_date, end_date);
```

```
                                break;
                            }
                            case 3: {
                                // Display available temperature range
                                displayAvailableTemperatureRange(data);

                                // Filter by temperature range
                                double min_temp, max_temp;
                                std::cout << "Enter minimum temperature: ";
                                std::cin >> min_temp;
                                std::cout << "Enter maximum temperature: ";
                                std::cin >> max_temp;

                                std::cout << "Filtering candlesticks...\n";
                                filtered_data =
filterByTemperatureRange(candlesticks, min_temp, max_temp);
                                break;
                            }
                            default:
                                std::cerr << "Invalid choice. Exiting
filtering...\n";
                                return 1;
                        }

                        // Plot the filtered data
                        if (!filtered_data.empty()) {
                            std::cout << "\nFiltered and Plotted Candlestick
Data:\n";
                            plotGroupedCandlesticks(filtered_data);
                        } else {
                            std::cout << "No data available for the selected
filter.\n";
                        }
                    }
                    break;
                }
```

This snippet of code provides an interface for users to interact with the filtering options. It captures user input for the desired filtering criteria and triggers the appropriate filtering and plotting functions. It calls functions (filterByCountry, filterByDateRange, or filterByTemperatureRange) based on the selected criteria. It also displays available filtering options (e.g., available countries, date range, or temperature range) to guide user input. Afterwards, it invokes plotGroupedCandlesticks to display the filtered results in a text-based format.

File “utils.cpp”:

```
// --- Task 3: Filtering Functions ---

/**
 * Filters candlesticks by a date range.
 *
 * @param candlesticks The list of candlestick data.
 * @param start_date The start date of the range (inclusive).
 * @param end_date The end date of the range (inclusive).
 * @return A vector of candlesticks within the specified date range.
 */
std::vector<Candlestick> filterByDateRange(
    const std::vector<Candlestick>& candlesticks,
    const std::string& start_date,
    const std::string& end_date) {
    std::vector<Candlestick> filtered;
    for (const auto& candle : candlesticks) {
        if (candle.date >= start_date && candle.date <= end_date) {
            filtered.push_back(candle);
        }
    }
    return filtered;
}

/**
 * Filters candlesticks by a temperature range.
 *
 * @param candlesticks The list of candlestick data.
 * @param min_temp The minimum temperature.
 * @param max_temp The maximum temperature.
 * @return A vector of candlesticks within the specified temperature range.
 */
std::vector<Candlestick> filterByTemperatureRange(
    const std::vector<Candlestick>& candlesticks,
    double min_temp,
    double max_temp) {
    std::vector<Candlestick> filtered;

    for (const auto& candle : candlesticks) {
        if (candle.high < min_temp || candle.low > max_temp) {
            continue;
        }

        // Create a truncated candlestick within the range
        Candlestick truncated = candle;
        truncated.high = std::min(candle.high, max_temp);
        truncated.low = std::max(candle.low, min_temp);
        truncated.open = std::clamp(candle.open, truncated.low,
truncated.high);
```

```
        truncated.close = std::clamp(candle.close, truncated.low,
truncated.high);
        filtered.push_back(truncated);
    }

    return filtered;
}

/**
 * Filters candlesticks by country and time frame.
 *
 * @param data The dataset as a 2D vector of strings.
 * @param country_prefix The country prefix (e.g., "AT" for Austria).
 * @param time_frame The time frame (e.g., "year", "month", or "day").
 * @return A vector of candlesticks for the specified country and time
frame.
 */
std::vector<Candlestick> filterByCountry(
    const std::vector<std::vector<std::string>>& data,
    const std::string& country_prefix,
    const std::string& time_frame) {
    try {
        return computeCandlestickData(data, country_prefix, time_frame);
    } catch (const std::exception& e) {
        std::cerr << "Error during country filtering: " << e.what() <<
std::endl;
        return {};
    }
}

/**
 * Provides a mapping of country prefixes to country names.
 *
 * @return A map where keys are country prefixes (e.g., "AT") and values are
country names.
 */
std::map<std::string, std::string> getCountryMapping() {
    return {
        {"AT", "Austria"}, {"BE", "Belgium"}, {"BG", "Bulgaria"}, {"CH",
"Switzerland"},
        {"CZ", "Czech Republic"}, {"DE", "Germany"}, {"DK", "Denmark"},
{"EE", "Estonia"},
        {"ES", "Spain"}, {"FI", "Finland"}, {"FR", "France"}, {"GB", "United
Kingdom"},
        {"GR", "Greece"}, {"HR", "Croatia"}, {"HU", "Hungary"}, {"IE",
"Ireland"},
        {"IT", "Italy"}, {"LT", "Lithuania"}, {"LU", "Luxembourg"}, {"LV",
"Latvia"},
        {"NL", "Netherlands"}, {"NO", "Norway"}, {"PL", "Poland"}, {"PT",
"Portugal"},
```

```
        {"RO", "Romania"}, {"SE", "Sweden"}, {"SI", "Slovenia"}, {"SK",
"Slovakia"}
    };
}

/**
 * Displays the available countries for filtering.
 *
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableCountries(const std::vector<std::vector<std::string>>&
data) {
    auto country_map = getCountryMapping();

    std::cout << "\n--- Available Country Prefixes and Names ---\n";
    for (const auto& header : data[0]) {
        if (header.find("_temperature") != std::string::npos) {
            std::string country_prefix = header.substr(0, header.find("_"));
            std::string country_name = country_map.count(country_prefix) > 0
                ? country_map[country_prefix]
                : "Unknown";
            std::cout << "- " << country_prefix << " (" << country_name <<
")\n";
        }
    }
    std::cout << std::endl;
}

/**
 * Displays the temperature range available in Europe.
 *
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableTemperatureRange(const
std::vector<std::vector<std::string>>& data) {
    double global_min_temp = std::numeric_limits<double>::max();
    double global_max_temp = std::numeric_limits<double>::lowest();

    for (size_t i = 0; i < data[0].size(); ++i) {
        if (data[0][i].find("_temperature") != std::string::npos) {
            for (size_t j = 1; j < data.size(); ++j) {
                try {
                    double temp = std::stod(data[j][i]);
                    global_min_temp = std::min(global_min_temp, temp);
                    global_max_temp = std::max(global_max_temp, temp);
                } catch (const std::exception&) {
                    continue;
                }
            }
        }
```

```
    }

    if (global_min_temp != std::numeric_limits<double>::max() &&
        global_max_temp != std::numeric_limits<double>::lowest()) {
        std::cout << "\n--- Global Temperature Range ---\n";
        std::cout << "Minimum: " << global_min_temp << " degree Celsius\n";
        std::cout << "Maximum: " << global_max_temp << " degree Celsius\n";
    } else {
        std::cout << "No valid temperature data found.\n";
    }
}

/**
 * Displays the available date range.
 *
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableDateRange(const std::vector<std::vector<std::string>>&
data) {
    if (data.size() < 2) {
        std::cout << "No date range available (data might be empty).\n";
        return;
    }

    std::string start_date = data[1][0];
    std::string end_date = data[data.size() - 1][0];

    std::cout << "\n--- Available Date Range ---\n";
    std::cout << "Start: " << start_date.substr(0, 10) << "\n";
    std::cout << "End: " << end_date.substr(0, 10) << "\n\n";
}
```

It implements the filtering logic for each user-specified criterion and provides utility functions to retrieve and display filtering options. The functions: filterByCountry, filterByDateRange, and filterByTemperatureRange, enable users to refine candlestick data based on specific criteria: country prefix, date range, or temperature range.

File “utils.h”:

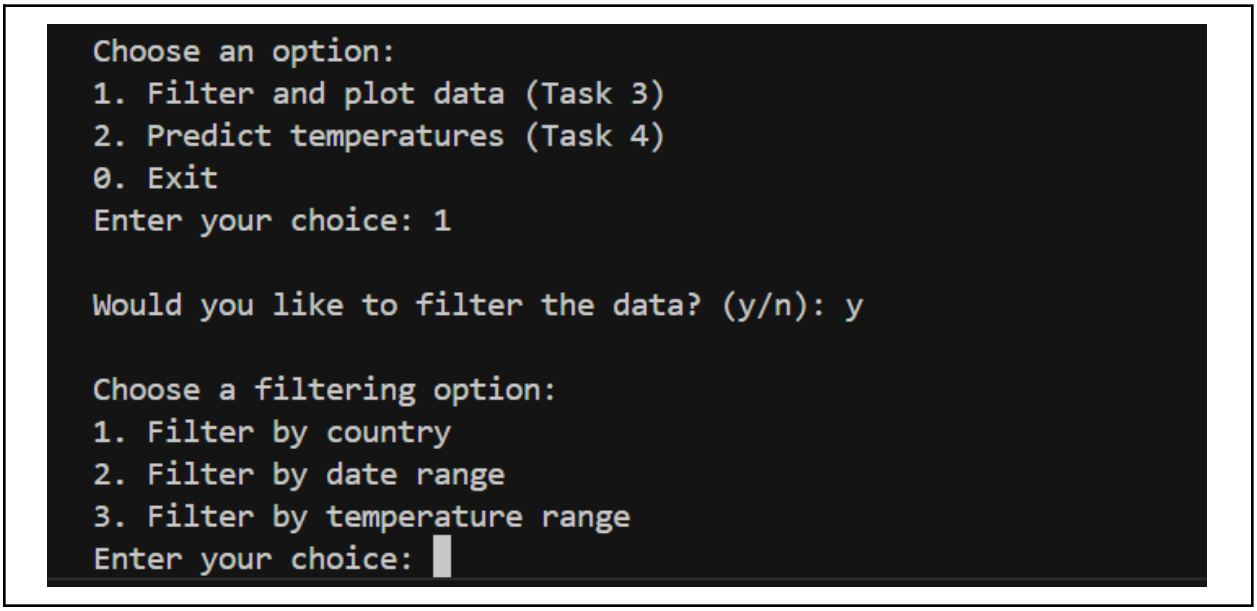
```
// --- Task 3: Filtering Functions ---

/**
 * Filters candlestick data by a specified date range.
 *
 * @param candlesticks A vector of Candlestick objects to filter.
 * @param start_date The start date of the range.
 * @param end_date The end date of the range.
 * @return A vector of filtered Candlestick objects.
 */
std::vector<Candlestick> filterByDateRange(
    const std::vector<Candlestick>& candlesticks,
    const std::string& start_date,
    const std::string& end_date
);

/**
 * Filters candlestick data by a specified temperature range.
 *
 * @param candlesticks A vector of Candlestick objects to filter.
 * @param min_temp The minimum temperature.
 * @param max_temp The maximum temperature.
 * @return A vector of filtered Candlestick objects.
 */
std::vector<Candlestick> filterByTemperatureRange(
    const std::vector<Candlestick>& candlesticks,
    double min_temp,
    double max_temp
);

/**
 * Filters by a specific country and time frame.
 *
 * @param data The dataset as a 2D vector of strings.
 * @param country_prefix The country prefix (e.g., "AT" for Austria).
 * @param time_frame The time frame (e.g., "year", "month", or "day").
 * @return A vector of filtered Candlestick objects.
 */
std::vector<Candlestick> filterByCountry(
    const std::vector<std::vector<std::string>>& data,
    const std::string& country_prefix,
    const std::string& time_frame
);

// Display Filter Options
/**
 * Displays available countries in the dataset.
 *
```

```
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableCountries(const std::vector<std::vector<std::string>>&
data);

/**
 * Displays the available date range in the dataset.
 *
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableDateRange(const std::vector<std::vector<std::string>>&
data);

/**
 * Displays the available temperature range in the dataset.
 *
 * @param data The dataset as a 2D vector of strings.
 */
void displayAvailableTemperatureRange(const
std::vector<std::vector<std::string>>& data);
```

This snippet of code declares functions for filtering candlestick data and retrieving filter options. It provides a clear interface for the filtering functionality, ensuring that other files can access and utilize these functions efficiently.

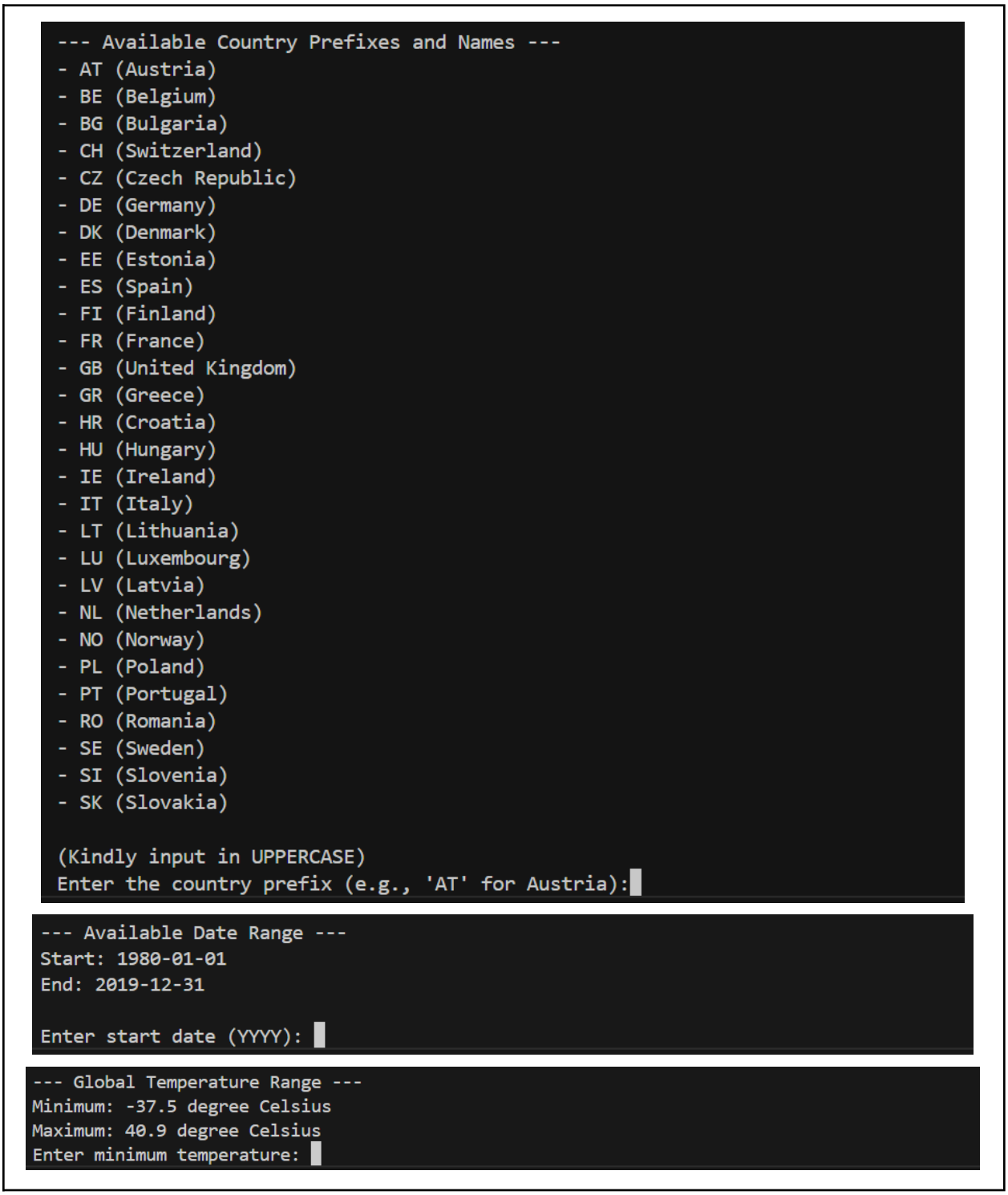
Output for Task 3:

Filtering Options



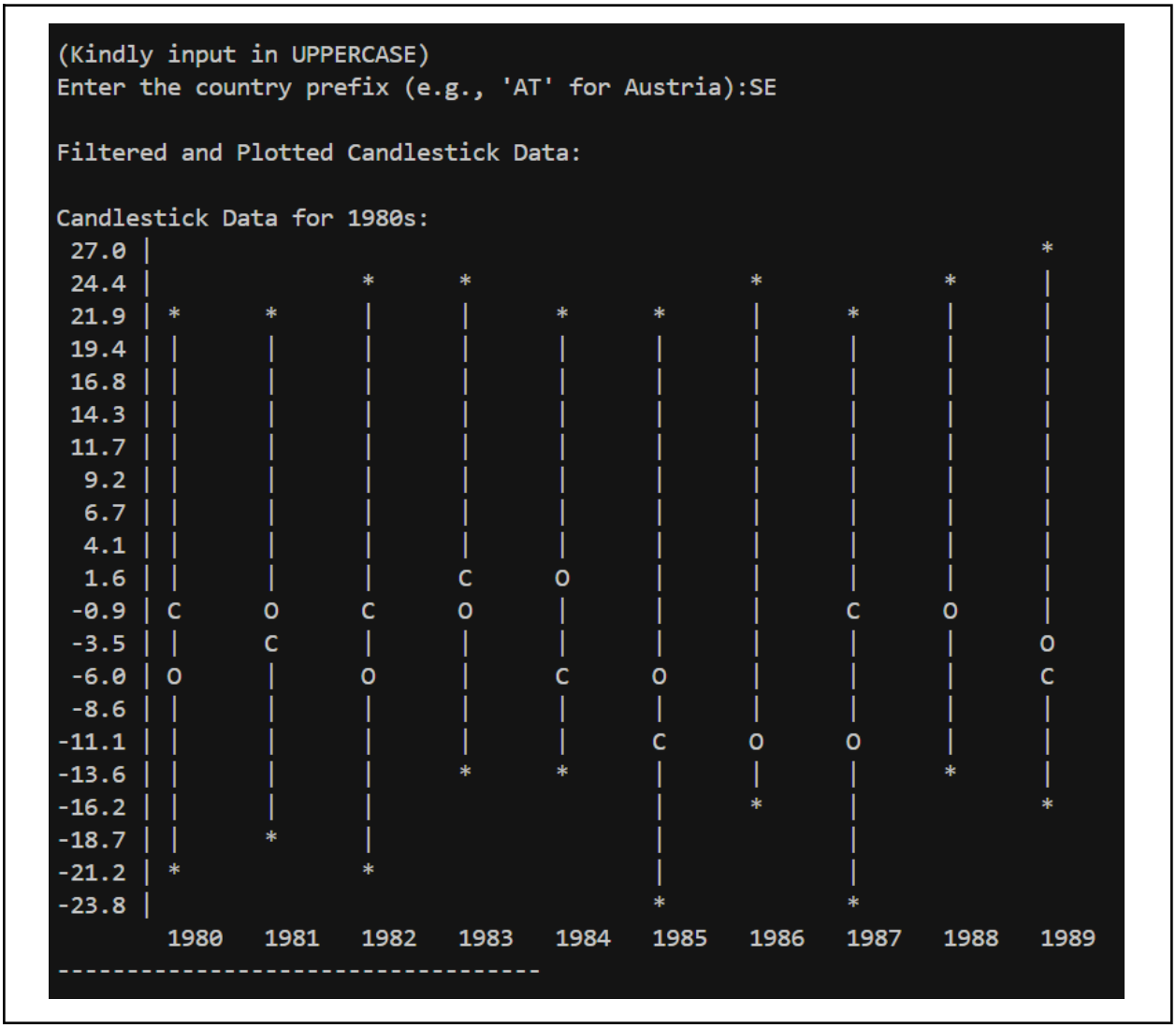
This output presents the main menu for Task 3, prompting users to choose a filtering option. The user is asked whether they want to filter the data and are allowed to choose from three filtering criteria: country, date range, or temperature range.

Listing available filtering options



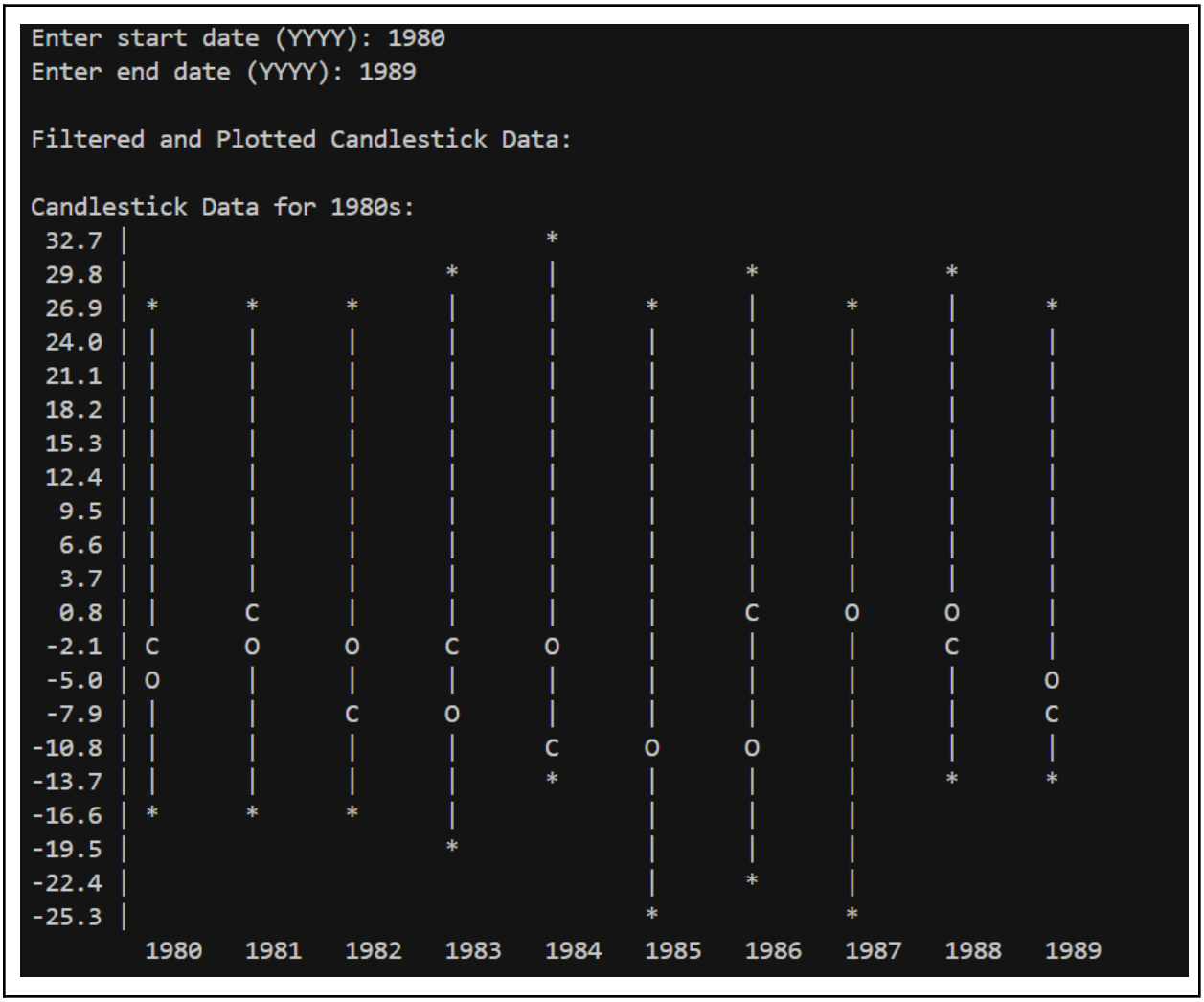
This displays available country prefixes, date ranges, and temperature ranges for filtering. It provides users with necessary information about the dataset, such as countries included, the earliest and latest dates, and the temperature range.

Filter by Country



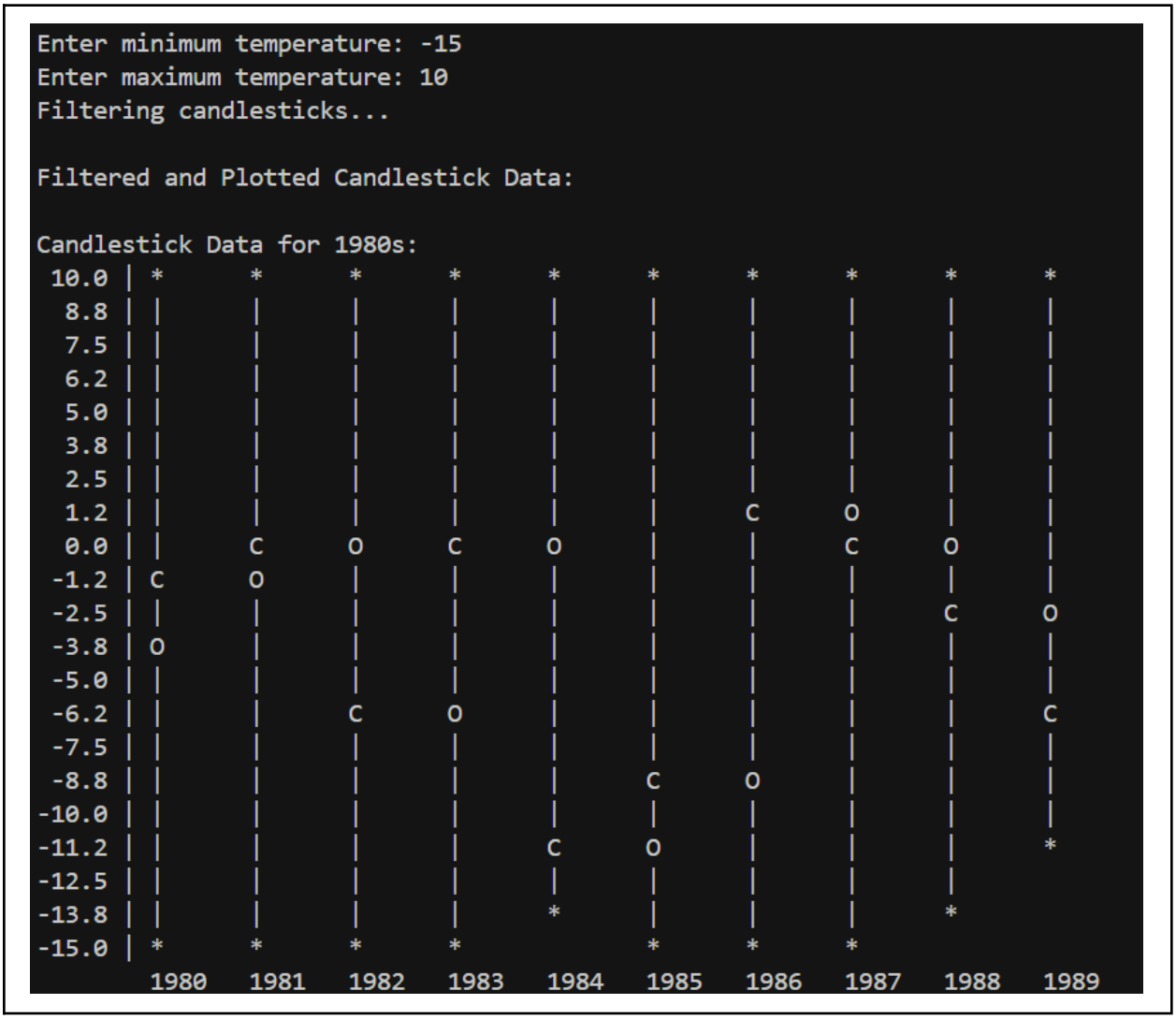
This output shows the filtered candlestick data for a specific country (e.g., Sweden, with prefix "SE") from 1980 to 2019, grouped by decades, highlighting trends for that specific country in a text-based plot.

Filter by Date Range



This output demonstrates filtering based on a user-specified date range (e.g., 1980–1989). The filtered candlestick data is grouped and plotted by decade, providing a detailed view of temperature trends within the selected timeframe.

Filter by Temperature Range



This output illustrates filtering by a specified temperature range (e.g., -15°C to 10°C). The filtered data is plotted, showing only the candlestick data within the given temperature limits, enabling focused trend analysis for specific temperature ranges.

Challenges and Solutions

1. Challenge 1: Ensuring accurate filtering across large datasets.

It was addressed by developing robust validation within filtering functions to handle various edge cases, such as invalid inputs, missing data, and out-of-range values, ensuring reliable and precise filtering.

2. Challenge 2: Handling empty results after filtering.

It was addressed by adding checks to detect cases where no data matches the selected filter criteria and providing user-friendly error messages to inform users effectively.

Key Insights

1. Improved Usability: Dynamic filtering options allow users to tailor the dataset to their specific needs, enabling focused analysis of particular countries, time periods, or temperature ranges.

2. Error Resilience: Validating inputs and handling edge cases enhances the program's robustness and user experience.

Task 4: Predicting Data and Plotting

Description:

In this task, we predict the average temperatures for a selected country and date range using historical candlestick data. The prediction is achieved through polynomial regression to identify patterns in historical data and forecast future averages. Results are visualized as a text-based plot, providing a clear interpretation of trends.

Implementation:

A polynomial regression is employed to analyze historical temperature trends. This model captures non-linear patterns more effectively than simple linear regression, allowing for nuanced predictions of future averages.

Data Filtering:

The prediction function filters data by country and date range to ensure that only relevant historical records are considered. The candlestick data is processed to extract years and average temperatures for input into the regression model.

Visualization:

The prediction results and historical data are plotted as a text-based graph for better user interpretation.

The graph includes:

- A Y-axis dynamically scaled to temperature ranges.
- Markers for historical data (O) and predicted data (*) to distinguish actual and forecasted points.
- A legend clarifying the symbols and trends.

File: “main.cpp”

```
// --- Task 4: Predictive Modelling ---

                case 2: {
                    // Predict temperatures
                    std::cout << "\nTask 4: Predicting Temperatures\n";
                    // Display available countries
                    displayAvailableCountries(data);

                    // Prompt user to select country
                    std::string country_prefix;
                    std::cout << "(Kindly input in UPPERCASE)\n";
                    std::cout << "Enter country prefix for prediction (e.g.,
'AT' for Austria):";
                    std::cin >> country_prefix;

                    // Prompt user for start and end years
                    int startYear, endYear;
                    std::cout << "Enter start year for prediction: ";
                    std::cin >> startYear;
                    std::cout << "Enter end year for prediction: ";
                    std::cin >> endYear;

                    // Perform prediction
                    predictAndDisplayTemperatures(data, country_prefix,
startYear, endYear);
                    break;
                }

                case 0:
                    std::cout << "Exiting program.\n";
                    proceed = 'n';
                    break;
                default:
                    std::cerr << "Invalid choice. Please try again.\n";
                    break;
            }

            if (choice != 0) {
                std::cout << "\nWould you like to perform another task?
(y/n): ";
                std::cin >> proceed;
            }
        } while (proceed == 'y' || proceed == 'Y');

    } catch (const std::exception& e) {
        // Handle any errors during computation or plotting
        std::cerr << "An error occurred: " << e.what() << std::endl;
        return 1; // Exit with error code
```

```
    }
    return 0; // Program Exit Successfully
}
```

This snippet of code handles user input for selecting a country and date range for prediction. It calls predictAndDisplayTemperatures() to perform data filtering, regression, and visualization. It includes error handling for invalid input or data unavailability.

FIle: “utils.cpp”

```
// Task 4: Polynomial Regression

/**
 * Performs polynomial regression to fit a polynomial to the given data
points
 * and predicts values for specified x-coordinates.
 *
 * @param x A vector of x-coordinates (independent variable, e.g., years).
 * @param y A vector of y-coordinates (dependent variable, e.g.,
temperatures).
 * @param degree The degree of the polynomial to fit.
 * @param predict_x A vector of x-coordinates for which predictions are
needed.
 * @return A vector of predicted y-coordinates corresponding to predict_x.
 */
std::vector<double> polynomialRegression(const std::vector<int>& x,
                                         const std::vector<double>& y,
                                         int degree,
                                         const std::vector<int>& predict_x)
{
    int n = x.size(); // Number of data points
    int m = degree + 1; // Degree of the polynomial + 1 (number of
coefficients)

    // Create matrix for least squares
    std::vector<std::vector<double>> X(m, std::vector<double>(m, 0));
    std::vector<double> Y(m, 0);

    // Populate the elements of X and Y
    for (int i = 0; i < n; ++i) {
        double xi = 1.0; // Start with x^0
        for (int j = 0; j < m; ++j) {
            for (int k = 0; k < m; ++k) {
                X[j][k] += xi * std::pow(x[i], k); // Compute sum of powers
of x
            }
            Y[j] += xi * y[i]; // Compute sum of x*y
            xi *= x[i]; // Increment the power of x
        }
    }

    // Solve the linear system X * B = Y to find coefficients B
    std::vector<double> B(m, 0); // Coefficients of the polynomial
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < m; ++j) {
            if (i != j) {
                double ratio = X[j][i] / X[i][i]; // Ratio for elimination
                for (int k = 0; k < m; ++k) {
```

```
                    X[j][k] -= ratio * X[i][k]; // Eliminate element in the
row
                }
                Y[j] -= ratio * Y[i]; // Adjust corresponding element in Y
            }
        }
    }
    for (int i = 0; i < m; ++i) {
        B[i] = Y[i] / X[i][i]; // Calculate coefficients
    }

    // Use coefficients B to calculate predictions
    std::vector<double> predictions;
    for (const auto& px : predict_x) {
        double pred = 0.0; // Initialize prediction
        double xi = 1.0; // Start with x^0
        for (int j = 0; j < m; ++j) {
            pred += B[j] * xi; // Add term to prediction
            xi *= px; // Increment power of x
        }
        predictions.push_back(pred); // Store prediction
    }

    return predictions; // Return all predictions
}

/**
 * Predicts and displays temperature trends for a selected country based on
historical data.
 *
 * @param data A 2D vector of strings representing the dataset.
 * @param country_prefix The prefix for the country.
 * @param startYear The start year of the analysis period.
 * @param endYear The end year of the analysis period.
 */
void predictAndDisplayTemperatures(const
std::vector<std::vector<std::string>>& data,
                                   const std::string& country_prefix,
                                   int startYear, int endYear) {
    // Compute candlestick data for the selected country
    auto candlesticks = computeCandlestickData(data, country_prefix,
"year");

    // Extract years and average temperatures
    std::vector<int> years; // To store years
    std::vector<double> avg_temps; // To store average temperatures

    for (const auto& candle : candlesticks) {
        int year = std::stoi(candle.date); // Convert date string to year
        if (year >= startYear && year <= endYear) { // Filter by year range
```

```
            years.push_back(year); // Add year to list
            avg_temps.push_back((candle.high + candle.low) / 2); // Compute
average temperature
        }
    }

    // Check if data is available
    if (years.empty() || avg_temps.empty()) {
        std::cout << "No data available for the selected country and date
range.\n";
        return;
    }

    // Define prediction years
    std::vector<int> predict_years = {years.back() + 1, years.back() + 2,
years.back() + 3};

    // Perform polynomial regression to predict temperatures
    auto predictions = polynomialRegression(years, avg_temps, 2,
predict_years); // Degree 2 polynomial

    // Display historical data
    std::cout << "\n--- Historical Temperature Data ---\n";
    for (size_t i = 0; i < years.size(); ++i) {
        std::cout << "Year: " << years[i] << ", Avg Temp: " << avg_temps[i]
<< " degree Celsius\n";
    }

    // Display predictions
    std::cout << "\n--- Prediction Summary ---\n";
    std::cout << "Country: " << country_prefix << "\n";
    std::cout << "Date Range: " << startYear << " to " << endYear << "\n";
    std::cout << "Predicted Temperatures for Upcoming Years:\n";
    for (size_t i = 0; i < predict_years.size(); ++i) {
        std::cout << "Year: " << predict_years[i] << ", Predicted Temp: " <<
predictions[i] << " degree Celsius\n";
    }

    // Visualization: Text-Based Plot
    // Calculate the minimum and maximum temperatures
    double min_temp = *std::min_element(avg_temps.begin(), avg_temps.end());
    double max_temp = *std::max_element(avg_temps.begin(), avg_temps.end());
    min_temp = std::min(min_temp, *std::min_element(predictions.begin(),
predictions.end()));
    max_temp = std::max(max_temp, *std::max_element(predictions.begin(),
predictions.end()));

    // Calculate the range and plot height
    double range = max_temp - min_temp;
    int plot_height = 8; // Height of the text-based plot
```

```
    if (range == 0) range = 1; // Prevent division by zero

    std::cout << "\n--- Text-Based Visualization ---\n";

    // Determine the year interval based on the time period
    int time_period = years.back() - years.front() + 1;
    int year_interval = (time_period > 20) ? 5 : (time_period > 10 ? 2 : 1);

    // Configure plot alignment
    int column_width = 2; // Width for year labels
    int axis_spacing = 1; // Space between Y-axis and plot

    // Print the Y-axis and data points
for (int i = plot_height; i >= 0; --i) {
    double temp_level = min_temp + (i * range / plot_height); // Temperature
for this level
    std::cout << std::fixed << std::setprecision(1) << std::setw(6) <<
temp_level << " |";
    std::cout << std::string(axis_spacing, ' '); // Add space after Y-axis

    // Plot historical data for labelled years only
    for (size_t j = 0; j < years.size(); ++j) {
        if (years[j] % year_interval == 0) {
            double pos = (avg_temps[j] - min_temp) / range * plot_height;
            if (static_cast<int>(std::round(pos)) == i) {
                std::cout << std::setw(column_width - 1) << "O"; // Mark
historical data point
            } else {
                std::cout << std::string(column_width, ' '); // Maintain
spacing
            }
        } else {
            std::cout << std::string(column_width, ' '); // Maintain spacing
for unlabelled years
        }
    }

    // Plot predicted data for labelled years only
    for (size_t j = 0; j < predict_years.size(); ++j) {
        if (predict_years[j] % year_interval == 0) {
            double pos = (predictions[j] - min_temp) / range * plot_height;
            if (static_cast<int>(std::round(pos)) == i) {
                std::cout << std::setw(column_width - 1) << "*"; // Mark
predicted data point
            } else {
                std::cout << std::string(column_width, ' '); // Maintain
spacing
            }
        } else {
            std::cout << std::string(column_width, ' '); // Maintain spacing
```

```
for unlabelled years
        }
    }
    std::cout << "\n";
}

    // Print X-axis labels
    std::cout << "       "; // Align with Y-axis
    std::cout << std::string(axis_spacing, ' '); // Space after Y-axis

    // Print historical year labels
    for (size_t j = 0; j < years.size(); ++j) {
        if (years[j] % year_interval == 0) {
            std::cout << " '" << std::setw(2) << std::setfill('0') <<
(years[j] % 100);
        } else {
            std::cout << std::string(column_width, ' ');
        }
    }

    // Print predicted year labels
    for (size_t j = 0; j < predict_years.size(); ++j) {
        if (predict_years[j] % year_interval == 0) {
            std::cout << " '" << std::setw(2) << std::setfill('0') <<
(predict_years[j] % 100);
        } else {
            std::cout << std::string(column_width, ' ');
        }
    }
    std::cout << "\n";
}
```

The predictAndDisplayTemperatures function filters historical candlestick data based on the user's selected country and date range. Afterwards, a polynomial regression model is computed to fit historical data and predicts values for future years.

File: “utils.h”

```
// --- Task 4: Polynomial Regression ---

/**
 * Performs polynomial regression to predict values.
 *
 * @param x A vector of x-values (e.g., years).
 * @param y A vector of y-values (e.g., temperatures).
 * @param degree The degree of the polynomial to fit.
 * @param predict_x A vector of x-values for which predictions are made.
 * @return A vector of predicted y-values.
 */
std::vector<double> polynomialRegression(
    const std::vector<int>& x,
    const std::vector<double>& y,
    int degree,
    const std::vector<int>& predict_x
);

/**
 * Predicts and displays temperature trends for a given country and date
range.
 *
 * @param data The dataset as a 2D vector of strings.
 * @param country_prefix The country prefix.
 * @param startYear The start year for the prediction.
 * @param endYear The end year for the prediction.
 */
void predictAndDisplayTemperatures(
    const std::vector<std::vector<std::string>>& data,
    const std::string& country_prefix,
    int startYear,
    int endYear
);
```

It declares the polynomialRegression function for fitting the model and predicting values. It also declares predictAndDisplayTemperatures for filtering data, performing predictions, and visualizing results.

Output for Task 4:

For Country: AT (Austria) and Date Range: 2000 to 2010:

```
(Kindly input in UPPERCASE)
Enter country prefix for prediction (e.g., 'AT' for Austria):AT
Enter start year for prediction: 2000
Enter end year for prediction: 2010
```

```
 --- Historical Temperature Data ---
 Year: 2000, Avg Temp: 8.1 degree Celsius
 Year: 2001, Avg Temp: 6.4 degree Celsius
 Year: 2002, Avg Temp: 6.9 degree Celsius
 Year: 2003, Avg Temp: 8.3 degree Celsius
 Year: 2004, Avg Temp: 7.0 degree Celsius
 Year: 2005, Avg Temp: 6.1 degree Celsius
 Year: 2006, Avg Temp: 4.5 degree Celsius
 Year: 2007, Avg Temp: 11.1 degree Celsius
 Year: 2008, Avg Temp: 8.8 degree Celsius
 Year: 2009, Avg Temp: 7.2 degree Celsius
 Year: 2010, Avg Temp: 6.1 degree Celsius

 --- Prediction Summary ---
 Country: AT
 Date Range: 2000 to 2010
 Predicted Temperatures for Upcoming Years:
 Year: 2011, Predicted Temp: 7.4 degree Celsius
 Year: 2012, Predicted Temp: 7.3 degree Celsius
 Year: 2013, Predicted Temp: 7.3 degree Celsius
```

```
--- Text-Based Visualization ---
  11.1 |
  10.2 |
   9.4 |
   8.6 |                 O
   7.8 | O
   7.0 |     O  O              *
   6.2 |                    O
   5.3 |
   4.5 |             O
         '00   '02   '04   '06   '08   '10   '12
```

For a selected country (e.g., AT) and date range (e.g., 2000 to 2010), the output displays average temperature trends for the specified years, and lists forecasted temperatures for future years, e.g., 2011, 2012, and 2013. The output also includes a text-based graph with a Y-axis representing temperature ranges. It differentiates between historical (O) and predicted (*) data points.

Challenges and Solutions

1. Challenge 1: Selecting a predictive model that balances complexity and accuracy.

Polynomial regression was implemented to capture non-linear trends in temperature data. This quadratic fit balances accuracy and computational efficiency, effectively modeling gradual and seasonal temperature variations.

2. Challenge 2: Visualizing the prediction in a user-friendly manner.

This was addressed by designing a text-based graph with clear legends for historical data and predicted data. The visualization dynamically scales based on temperature ranges and includes a legend for clarity, ensuring accessibility for all users.

Key Insights

1. Effective Trend Analysis: Polynomial regression provides a framework for modeling non-linear temperature patterns, offering insights beyond what linear regression can achieve.

2. Accessible Visualization: The text-based graph enables users that have no access to advanced graphical tools, to make predictions and historical data easy to interpret.