```cpp
#include "Utils.h"
#include "Candlestick.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <map>
#include <algorithm>
#include <limits>
#include <cmath>

// --- General Utility Functions ---

/**
 * Reads a CSV file and returns its content as a 2D vector of strings.
 *
 * @param filename The name of the CSV file.
 * @return A 2D vector where each inner vector represents a row of the
file.
 */
std::vector<std::vector<std::string>> readCSV(const std::string
&filename) {
    std::vector<std::vector<std::string>> data;
    std::ifstream file(filename);

    if (!file.is_open()) {
        std::cerr << "Error: Could not open file " << filename <<
std::endl;
        return data;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::vector<std::string> row;
        std::stringstream ss(line);
        std::string cell;

        while (std::getline(ss, cell, ',')) {
            row.push_back(cell);
        }
```

```
        if (!row.empty()) {
            data.push_back(row);
        }
    }

    file.close();
    return data;
}

// --- 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;
}

// --- 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";
        }
    }
}

// --- 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";
}

// 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";
}
```