**Working C++ Files**

**ComputeCandlesticks.cpp**

*// ComputeCandlesticks.cpp*

#include "ComputeCandlesticks.h"

#include <map>

#include <vector>

#include <numeric>

#include <algorithm>

#include <iostream>

std::vector<Candlestick> computeCandlesticks(const std::vector<std::pair<std::string, double>>& entries) {

    std::map<std::string, std::vector<double>> groupedData;

*// Group temperatures by year (YYYY)*

    for (const auto& entry : entries) {

        if (entry.first.length() < 4) {

            std::cerr << "Warning: Invalid date format '" << entry.first << "'" << std::endl;

            continue;

        }

        std::string year = entry.first.substr(0, 4); *// Extract YYYY*

        groupedData[year].push\_back(entry.second);

    }

    std::vector<Candlestick> candlesticks;

    double prevClose = 0.000; *// Initialize to 0.000 for the year 1980*

*// Compute candlestick data for each group from 1980 to 2019*

    for (int year = 1980; year <= 2019; ++year) {

        std::string yearStr = std::to\_string(year);

        if (groupedData.find(yearStr) != groupedData.end()) {

            const std::vector<double>& temperatures = groupedData[yearStr];

            double open = prevClose;

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

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

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

            candlesticks.emplace\_back(yearStr, open, close, high, low);

            prevClose = close;

        } else {

            std::cerr << "Warning: Missing data for year " << yearStr << std::endl;

            candlesticks.emplace\_back(yearStr, prevClose, prevClose, prevClose, prevClose);

        }

    }

*// Debug output*

    std::cout << "Debug: Computed " << candlesticks.size() << " candlesticks.\n";

    for (const auto& candle : candlesticks) {

        std::cout << "Year: " << candle.date << ", Open: " << candle.open

                  << ", Close: " << candle.close << ", High: " << candle.high

                  << ", Low: " << candle.low << std::endl;

    }

    return candlesticks;

}

**CSVReader.cpp**

*// CSVReader.cpp*

#include "CSVReader.h"

*/\*Reads temperature data for the specified column (country) from a CSV file, filename Name of the CSV file,*

*column Name of the temperature column (e.g., "GB\_temperature") and Vector of pairs (year, temperature)*

*\*/*

std::vector<std::pair<std::string, double>> CSVReader::readCSV(const std::string& filename, const std::string& column) {

    std::ifstream file(filename);

    if (!file.is\_open()) {

        throw std::runtime\_error("Unable to open file: " + filename);

    }

    std::string line, header;

    std::getline(file, header); *// Read header*

*// Find the column index*

    std::istringstream headerStream(header);

    std::string cell;

    int colIndex = -1, idx = 0;

    while (std::getline(headerStream, cell, ',')) {

        if (cell == column) {

            colIndex = idx;

            break;

        }

        idx++;

    }

    if (colIndex == -1) {

        throw std::runtime\_error("Column not found: " + column);

    }

*// Read data*

    std::vector<std::pair<std::string, double>> data;

    while (std::getline(file, line)) {

        std::istringstream lineStream(line);

        std::string timestamp, value;

        int col = 0;

        while (std::getline(lineStream, cell, ',')) {

            if (col == 0) {

*// Extract year from "1980-01-01T00:00:00Z"*

                if (cell.length() < 4) {

                    std::cerr << "Warning: Invalid date format '" << cell << "'" << std::endl;

                    timestamp = "Unknown";

                } else {

                    timestamp = cell.substr(0, 4); *// Year*

                }

            } else if (col == colIndex) {

                value = cell;

            }

            col++;

        }

        if (!value.empty() && timestamp != "Unknown") {

            try {

                data.emplace\_back(timestamp, std::stod(value));

            } catch (const std::invalid\_argument&) {

                std::cerr << "Warning: Invalid temperature value '" << value << "' on date " << timestamp << std::endl;

            }

        }

    }

    file.close();

*// Debug output*

    std::cout << "Debug: Extracted " << data.size() << " rows of data from CSV.\n";

    return data;

}

**DataFilter.cpp**

*// DataFilter.cpp*

#include "DataFilter.h"

#include <algorithm>

*//  Filters candlesticks based on a year range.*

std::vector<Candlestick> filterByYearRange(const std::vector<Candlestick>& candlesticks, int startYear, int endYear) {

    std::vector<Candlestick> filtered;

    for (const auto& candle : candlesticks) {

        int year = std::stoi(candle.date);

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

            filtered.push\_back(candle);

        }

    }

    return filtered;

}

*// Filters candlesticks based on a closing temperature range.*

std::vector<Candlestick> filterByClosingTemperatureRange(const std::vector<Candlestick>& candlesticks, double minTemp, double maxTemp) {

    std::vector<Candlestick> filtered;

    for (const auto& candle : candlesticks) {

        if (candle.close >= minTemp && candle.close <= maxTemp) {

            filtered.push\_back(candle);

        }

    }

    return filtered;

}

**Main.cpp**

*// main.cpp*

    #include "Candlestick.h"

    #include "CSVReader.h"

    #include "ComputeCandlesticks.h"

    #include "DataFilter.h"

    #include "PlotCandlesticks.h"

    #include "TemperaturePredictor.h"

    #include <iostream>

    #include <vector>

    #include <string>

    #include <fstream>

    #include <sstream>

    #include <map>

    #include <iomanip>

    #include <limits>

    std::vector<std::string> getColumnNames(const std::string& filename) {

        std::ifstream file(filename);

        std::vector<std::string> columnNames;

        if (file.is\_open()) {

            std::string headerLine;

            std::getline(file, headerLine); *// Read the first line*

            std::stringstream ss(headerLine);

            std::string column;

            while (std::getline(ss, column, ',')) {

                columnNames.push\_back(column);

            }

        } else {

            throw std::runtime\_error("Unable to open file: " + filename);

        }

        return columnNames;

    }

*//  Dynamically extracts country columns from the dataset.*

*//  Vector of column names from the dataset.*

*//   Map of menu index to column name (e.g., "AT\_temperature").*

    std::map<int, std::string> extractCountryColumns(const std::vector<std::string>& columnNames) {

        std::map<int, std::string> countryMenu;

        int index = 1;

        for (const auto& column : columnNames) {

*// Only include columns ending with "\_temperature"*

            if (column.find("\_temperature") != std::string::npos) {

                countryMenu[index++] = column;

            }

        }

        if (countryMenu.empty()) {

            throw std::runtime\_error("No country temperature columns found in the CSV file.");

        }

        return countryMenu;

    }

*// Converts a column name into a user-friendly country name.*

*//  Column name from the dataset (e.g., "AT\_temperature").*

*//  Formatted country name (e.g., "Austria Temperature").*

    std::string formatCountryName(const std::string& columnName) {

        std::string countryCode = columnName.substr(0, columnName.find("\_"));

        std::string countryName = countryCode; *// Default to code if not mapped*

*// Example map of country codes to full names (add more if needed)*

        static std::map<std::string, std::string> countryNameMap = {

            {"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"}

        };

        if (countryNameMap.find(countryCode) != countryNameMap.end()) {

            countryName = countryNameMap[countryCode];

        }

        return countryName + " Temperature";

    }

    std::string selectCountry(const std::map<int, std::string>& countryMenu) {

        while (true) {

            std::cout << "\nAvailable Countries:\n";

            for (const auto& it : countryMenu) {

                std::cout << it.first << ". " << formatCountryName(it.second) << std::endl;

            }

            std::cout << "Enter the number of the country: ";

            int choice;

            std::cin >> choice;

            if (std::cin.fail()) {

                std::cin.clear(); *// Clear the error flags*

                std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); *// Discard invalid input*

                std::cout << "Invalid input. Please enter a valid number.\n";

                continue;

            }

            if (countryMenu.find(choice) != countryMenu.end()) {

                return countryMenu.at(choice);

            } else {

                std::cout << "Invalid selection. Please try again.\n";

            }

        }

    }

*// Displays candlestick data in a tabular format.*

    void displayCandlesticksAsTable(const std::vector<Candlestick>& candlesticks) {

*// Print header*

        std::cout << std::setw(15) << "Year"

                << std::setw(10) << "Open"

                << std::setw(10) << "High"

                << std::setw(10) << "Low"

                << std::setw(10) << "Close" << std::endl;

*// Print separator line*

        std::cout << std::string(55, '-') << std::endl;

*// Print each candlestick row*

        for (const auto& candle : candlesticks) {

            std::cout << std::setw(15) << candle.date

                    << std::setw(10) << std::fixed << std::setprecision(3) << candle.open

                    << std::setw(10) << candle.high

                    << std::setw(10) << candle.low

                    << std::setw(10) << candle.close << std::endl;

        }

    }

*// Prompts the user for filter criteria and filters the candlestick data accordingly.*

    std::vector<Candlestick> filterCandlesticks(const std::vector<Candlestick>& candlesticks) {

        std::vector<Candlestick> filtered;

        bool filterDate = false, filterTemp = false;

        int startYear = 1980, endYear = 2019;

        double minTemp = -1000.0, maxTemp = 1000.0;

*// Ask user if they want to filter by date range*

        std::cout << "Do you want to filter by date range? (y/n): ";

        char choice;

        std::cin >> choice;

        if (choice == 'y' || choice == 'Y') {

            filterDate = true;

            std::cout << "Enter start year (1980-2019): ";

            std::cin >> startYear;

            std::cout << "Enter end year (1980-2019): ";

            std::cin >> endYear;

*// Validate years*

            if (startYear < 1980 || startYear > 2019 || endYear < 1980 || endYear > 2019 || startYear > endYear) {

                std::cerr << "Invalid year range. Please ensure 1980 <= start year <= end year <= 2019.\n";

*// Reset to default*

                startYear = 1980;

                endYear = 2019;

                filterDate = false;

            }

        }

*// Ask user if they want to filter by closing temperature range*

        std::cout << "Do you want to filter by closing temperature range? (y/n): ";

        std::cin >> choice;

        if (choice == 'y' || choice == 'Y') {

            filterTemp = true;

            std::cout << "Enter minimum closing temperature: ";

            std::cin >> minTemp;

            std::cout << "Enter maximum closing temperature: ";

            std::cin >> maxTemp;

*// Validate temperatures*

            if (minTemp > maxTemp) {

                std::cerr << "Invalid temperature range. Minimum temperature cannot be greater than maximum temperature.\n";

*// Reset to default*

                minTemp = -1000.0;

                maxTemp = 1000.0;

                filterTemp = false;

            }

        }

*// Apply filters using DataFilter functions*

        std::vector<Candlestick> tempFiltered = candlesticks;

        if (filterDate) {

            tempFiltered = filterByYearRange(tempFiltered, startYear, endYear);

        }

        if (filterTemp) {

            tempFiltered = filterByClosingTemperatureRange(tempFiltered, minTemp, maxTemp);

        }

        filtered = tempFiltered;

*// Debug output*

        std::cout << "Debug: Filtered " << filtered.size() << " candlesticks after applying filters.\n";

        return filtered;

    }

*// Handles the filtering and plotting functionality.*

    void filterAndPlot(const std::map<int, std::string>& countryMenu, const std::string& filename) {

        try {

*// Select Country*

            std::string selectedColumn = selectCountry(countryMenu);

            std::cout << "You selected: " << formatCountryName(selectedColumn) << std::endl;

*// Read CSV data for the selected country*

            std::vector<std::pair<std::string, double>> data = CSVReader::readCSV(filename, selectedColumn);

*// Compute Candlesticks*

            std::vector<Candlestick> candlesticks = computeCandlesticks(data);

*// Apply Filters*

            std::vector<Candlestick> filteredCandlesticks = filterCandlesticks(candlesticks);

            if (filteredCandlesticks.empty()) {

                std::cout << "No data matches the specified filters.\n";

                return;

            }

*// Display filtered candlesticks*

            std::cout << "\nFiltered Candlestick Data:\n";

            displayCandlesticksAsTable(filteredCandlesticks);

*// Plot filtered candlesticks*

            std::cout << "\nPlotting Filtered Candlestick Data:\n";

            plotCandlesticks(filteredCandlesticks, 20); *// Adjust scaleHeight as needed*

        } catch (const std::exception& e) {

            std::cerr << "Error during filtering and plotting: " << e.what() << std::endl;

        }

    }

    int main() {

        const std::string filename = "weather\_data.csv";

        std::vector<std::string> columnNames;

        std::map<int, std::string> countryMenu;

        std::vector<std::pair<std::string, double>> data;

        std::vector<Candlestick> candlesticks;

        std::string selectedColumn;

        try {

*// Extract column names from the CSV file*

            columnNames = getColumnNames(filename);

*// Dynamically extract country columns*

            countryMenu = extractCountryColumns(columnNames);

*// Menu Loop*

            while (true) {

*// Display Menu*

                std::cout << "\nWeather Analysis Menu\n";

                std::cout << "1. Select Country and Display Table\n";

                std::cout << "2. Plot Candlestick Data\n";

                std::cout << "3. Filter Plot Data\n";

                std::cout << "4. Predict Temperature Changes\n"; *// New option*

                std::cout << "5. Exit\n"; *// Updated Exit option number*

                std::cout << "Enter your choice: ";

                int choice;

                std::cin >> choice;

*// Clear cin fail state if any non-integer is entered*

                if (std::cin.fail()) {

                    std::cin.clear(); *// clear the error flags*

                    std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); *// discard invalid input*

                    std::cout << "Invalid input. Please enter a number between 1 and 5.\n";

                    continue;

                }

                switch (choice) {

                    case 1:

                        try {

                            selectedColumn = selectCountry(countryMenu);

                            std::cout << "You selected: " << formatCountryName(selectedColumn) << std::endl;

*// Read and compute candlesticks for the selected country*

                            data = CSVReader::readCSV(filename, selectedColumn);

                            candlesticks = computeCandlesticks(data);

*// Display computed candlesticks immediately (Tabular Format Only)*

                            std::cout << "\nCandlestick Data:\n";

                            displayCandlesticksAsTable(candlesticks);

                        } catch (const std::exception& e) {

                            std::cerr << "Error: " << e.what() << std::endl;

                        }

                        break;

                    case 2:

                        if (candlesticks.empty()) {

                            std::cout << "No candlestick data available. Please select a country first.\n";

                        } else {

                            std::cout << "\nPlotting Candlestick Data:\n";

                            plotCandlesticks(candlesticks, 20); *// Adjust scaleHeight as needed*

                        }

                        break;

                    case 3:

                        if (candlesticks.empty()) {

                            std::cout << "No candlestick data available. Please select a country first.\n";

                        } else {

                            filterAndPlot(countryMenu, filename);

                        }

                        break;

                    case 4:

                        if (candlesticks.empty()) {

                            std::cout << "No candlestick data available. Please select a country first.\n";

                        } else {

                            try {

*// Prompt user for prediction range*

                                int predStartYear, predEndYear;

                                std::cout << "Enter the start year for prediction (e.g., 2020): ";

                                std::cin >> predStartYear;

                                std::cout << "Enter the end year for prediction (e.g., 2025): ";

                                std::cin >> predEndYear;

                                if (predStartYear > predEndYear) {

                                    std::cerr << "Invalid range. Start year must be less than or equal to end year.\n";

                                    break;

                                }

*// Initialize TemperaturePredictor with historical data*

                                TemperaturePredictor predictor(candlesticks);

*// Predict temperatures*

                                std::vector<Candlestick> predictions = predictor.predictTemperatures(predStartYear, predEndYear);

*// Display predictions*

                                std::cout << "\nPredicted Candlestick Data:\n";

                                displayCandlesticksAsTable(predictions);

*// Plot predictions*

                                std::cout << "\nPlotting Predicted Candlestick Data:\n";

                                plotCandlesticks(predictions, 20);

                            } catch (const std::exception& e) {

                                std::cerr << "Error during prediction: " << e.what() << std::endl;

                            }

                        }

                        break;

                    case 5:

                        std::cout << "Exiting Weather Analysis...\n";

                        return 0;

                    default:

                        std::cout << "Invalid choice. Please try again.\n";

                        break;

                }

            }

        } catch (const std::exception& e) {

            std::cerr << "Error: " << e.what() << std::endl;

        }

        return 0;

    }

**Plotcandlesticks.cpp**

*// PlotCandlesticks.cpp*

#include "PlotCandlesticks.h"

#include "Utils.h" *// For clamp and normalize*

#include <iostream>

#include <iomanip>

#include <cmath>

#include <algorithm>

#include <limits>

*//Plots candlestick data in a text-based format with spacing aligned to x-axis labels.*

void plotCandlesticks(const std::vector<Candlestick>& candlesticks, int scaleHeight) {

    if (candlesticks.empty()) {

        std::cerr << "No candlestick data to plot." << std::endl;

        return;

    }

*// Determine global min and max temperatures*

    double globalMin = candlesticks[0].low;

    double globalMax = candlesticks[0].high;

    for (const auto& candle : candlesticks) {

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

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

    }

*// Adjust scaleHeight if necessary*

    scaleHeight = std::max(scaleHeight, 10);

*// Pagination setup*

    const int pageSize = 20;

    int totalCandlesticks = candlesticks.size();

    int totalPages = (totalCandlesticks + pageSize - 1) / pageSize;

    for (int currentPage = 1; currentPage <= totalPages; ++currentPage) {

        int startIdx = (currentPage - 1) \* pageSize;

        int endIdx = std::min(startIdx + pageSize, totalCandlesticks);

*// Subset of candlesticks for the current page*

        std::vector<Candlestick> subset(candlesticks.begin() + startIdx, candlesticks.begin() + endIdx);

*// Print the plot from top (max) to bottom (min)*

        for (int row = scaleHeight - 1; row >= 0; --row) {

            double currentTemp = globalMin + (globalMax - globalMin) \* row / (scaleHeight - 1);

            std::cout << std::setw(6) << std::fixed << std::setprecision(1) << currentTemp << " | ";

            for (const auto& candle : subset) {

                int highPos = normalize(candle.high, globalMin, globalMax, scaleHeight);

                int lowPos = normalize(candle.low, globalMin, globalMax, scaleHeight);

                int openPos = normalize(candle.open, globalMin, globalMax, scaleHeight);

                int closePos = normalize(candle.close, globalMin, globalMax, scaleHeight);

                if (row == highPos && row == lowPos) {

                    std::cout << "|    ";

                } else if (row == highPos) {

                    std::cout << "|    ";

                } else if (row == lowPos) {

                    std::cout << "|    ";

                } else if (row == openPos && row == closePos) {

                    std::cout << "=    ";

                } else if (row == openPos) {

                    std::cout << "+    ";

                } else if (row == closePos) {

                    std::cout << "-    ";

                } else if (row < highPos && row > lowPos) {

                    std::cout << "|    ";

                } else {

                    std::cout << "     "; *// Add appropriate spacing*

                }

            }

            std::cout << std::endl;

        }

*// Print the x-axis separator*

        std::cout << "       " << std::string(subset.size() \* 5, '-') << std::endl;

*// Print the years below the plot with spacing aligned*

        std::cout << "       ";

        for (const auto& candle : subset) {

            std::cout << std::setw(5) << candle.date.substr(0, 4) << " ";

        }

        std::cout << std::endl;

*// If there are more pages, prompt the user to continue*

        if (currentPage < totalPages) {

            std::cout << "\nPress Enter to view the next page...";

            std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); *// Clear input buffer*

            std::cin.get(); *// Wait for Enter*

        }

    }

}

**TemperaturePredictor.cpp**

*// TemperaturePredictor.cpp*

#include "TemperaturePredictor.h"

#include <numeric>

#include <cmath>

#include <iostream>

*/\*\**

*\* @brief Constructor that initializes the historical data.*

*\*/*

TemperaturePredictor::TemperaturePredictor(const std::vector<Candlestick>& historicalData) : data(historicalData) {}

*/\*\**

*\* @brief Calculates the slope (m) and intercept (c) for linear regression.*

*\*/*

void TemperaturePredictor::calculateLinearRegression(double& m, double& c) {

    int n = data.size();

    if (n == 0) {

        throw std::runtime\_error("No data available for prediction.");

    }

    std::vector<double> years;

    std::vector<double> temperatures;

    for (const auto& candle : data) {

        years.push\_back(std::stod(candle.date));

        temperatures.push\_back(candle.close);

    }

    double sum\_x = std::accumulate(years.begin(), years.end(), 0.0);

    double sum\_y = std::accumulate(temperatures.begin(), temperatures.end(), 0.0);

    double sum\_xy = 0.0;

    double sum\_x2 = 0.0;

    for (int i = 0; i < n; ++i) {

        sum\_xy += years[i] \* temperatures[i];

        sum\_x2 += years[i] \* years[i];

    }

    double denominator = n \* sum\_x2 - sum\_x \* sum\_x;

    if (denominator == 0) {

        throw std::runtime\_error("Denominator in linear regression calculation is zero.");

    }

    m = (n \* sum\_xy - sum\_x \* sum\_y) / denominator;

    c = (sum\_y \* sum\_x2 - sum\_x \* sum\_xy) / denominator;

}

*/\*\**

*\* @brief Predicts temperatures for a given range of years using linear regression.*

*\*/*

std::vector<Candlestick> TemperaturePredictor::predictTemperatures(int startYear, int endYear) {

    std::vector<Candlestick> predictions;

    double m, c;

    calculateLinearRegression(m, c);

    for (int year = startYear; year <= endYear; ++year) {

        double predictedClose = m \* year + c;

*// For simplicity, set open = close = predictedClose, high = predictedClose + 1, low = predictedClose - 1*

        double predictedOpen = predictedClose;

        double predictedHigh = predictedClose + 1.0;

        double predictedLow = predictedClose - 1.0;

        predictions.emplace\_back(std::to\_string(year), predictedOpen, predictedClose, predictedHigh, predictedLow);

    }

    return predictions;

}

**Header Files**

**Candlestick.h**

*// Candlestick.h*

#pragma once

#include <string>

class Candlestick {

public:

    std::string date; *// Year (e.g., "1980")*

    double open; *// Opening average temperature*

    double close; *// Closing average temperature*

    double high; *// Highest temperature*

    double low; *// Lowest temperature*

*// Constructor*

    Candlestick(const std::string& d, double o, double c, double h, double l)

        : date(d), open(o), close(c), high(h), low(l) {}

};

**ComputeCandlesticks.h**

*// ComputeCandlesticks.h*

#pragma once

#include "Candlestick.h"

#include <vector>

#include <string>

#include <utility>

*//Compute candlestick data for the specified time frame, Vector of pairs (year, temperature) and Vector of Candlestick objects*

std::vector<Candlestick> computeCandlesticks(const std::vector<std::pair<std::string, double>>& entries);

**CSVReader.h**

*// CSVReader.h*

#pragma once

#include <fstream>

#include <sstream>

#include <vector>

#include <string>

#include <stdexcept>

#include <iostream> *// For debug logs*

*// CSVReader class to read temperature data from CSV files*

class CSVReader {

public:

    static std::vector<std::pair<std::string, double>> readCSV(const std::string& filename, const std::string& column);

};

**DataFilter.h**

*// DataFilter.h*

#pragma once

#include "Candlestick.h"

#include <vector>

#include <string>

*// Filtering by the range of the year from 1980 to 2019*

std::vector<Candlestick> filterByYearRange(const std::vector<Candlestick>& candlesticks, int startYear, int endYear);

*// Filtering by the opening and closing temperature*

std::vector<Candlestick> filterByClosingTemperatureRange(const std::vector<Candlestick>& candlesticks, double minTemp, double maxTemp);

**PlotCandlesticks.h**

*// PlotCandlesticks.h*

#pragma once

#include "Candlestick.h"

#include <vector>

*// Text Based Plot*

void plotCandlesticks(const std::vector<Candlestick>& candlesticks, int scaleHeight);

**TemperaturePredictor.h**

*// TemperaturePredictor.h*

#pragma once

#include "Candlestick.h"

#include <vector>

#include <string>

#include <utility>

*// TemperaturePredictor class to predict future temperatures based on historical data.*

class TemperaturePredictor {

public:

*// Constructor that takes historical candlestick data.*

    TemperaturePredictor(const std::vector<Candlestick>& historicalData);

*// Predicts temperatures for a given range of years.*

    std::vector<Candlestick> predictTemperatures(int startYear, int endYear);

private:

    std::vector<Candlestick> data;

*//  Calculates the slope (m) and intercept (c) for linear regression.*

    void calculateLinearRegression(double& m, double& c);

};

**Utils.h**

*// Utils.h*

#pragma once

#include <cmath>

#include <algorithm>

*//Custom implementation of clamp for C++11.*

inline int clamp(int value, int minValue, int maxValue) {

    if (value < minValue) return minValue;

    if (value > maxValue) return maxValue;

    return value;

}

*//Normalize a value to fit within the scale height.*

inline int normalize(double value, double minValue, double maxValue, int scaleHeight) {

    if (maxValue == minValue) return 0; *// Prevent division by zero*

    double normalized = (value - minValue) / (maxValue - minValue) \* (scaleHeight - 1);

    return clamp(static\_cast<int>(std::round(normalized)), 0, scaleHeight - 1);

}