**Add New City**

**Purpose:** The purpose of this function is to **add a new city to the city list**.

**Prototype:** void addCity(char cities[][50], int\*count);

**Parameters:**

* cities[][50] – A 2D character array used to store city names.
* \*count – A pointer to the current number of cities in the list.

**How It Works:**

1. **Check capacity**
   * First, it checks if the city list is full (\*count >= MAX\_CITIES).
   * If full, it displays the message: “City list is full!”.
2. **Input city name**
   * It takes the city name from the user and stores it in the name array.
3. **Check uniqueness**
   * Compares the new name with existing cities in the list.
   * If any character does not match, notUnique = 0.
   * If the name matches an existing city, unique = 0.
4. **Add city**
   * If the name is unique, it copies the name into cities[\*count].
   * Increments the city count (\*count)++.
   * Displays: “City added successfully!”.
5. **Duplicate case**
   * If the name already exists in the list, it displays: “City name already exists!”.

A computer screen shot of a black screen

Description automatically generated

**Display all cities**

**Purpose:**  
The purpose of this function is to **display all the cities currently stored in the city list** in a numbered format.

**Prototype:**

void displayCities(char cities[][50], int count);

**Parameters:**

* cities[][50] – A 2D character array containing city names.
* count – The current number of cities in the list.

**How It Works:**

1. **Check if list is empty**
   * If count is 0, it displays: “No cities to display.”
2. **Display cities**
   * Iterates through all cities from index 0 to count - 1.
   * Prints each city with a number (1, 2, 3, …) for clarity.
   * Uses a while loop with putchar to print each character of the city name.

**Example Output:**

A computer screen with a black screen

Description automatically generated

### ****Function: renameCity****

**Purpose:**  
The purpose of this function is to **rename an existing city in the city list** while ensuring the new name is unique.

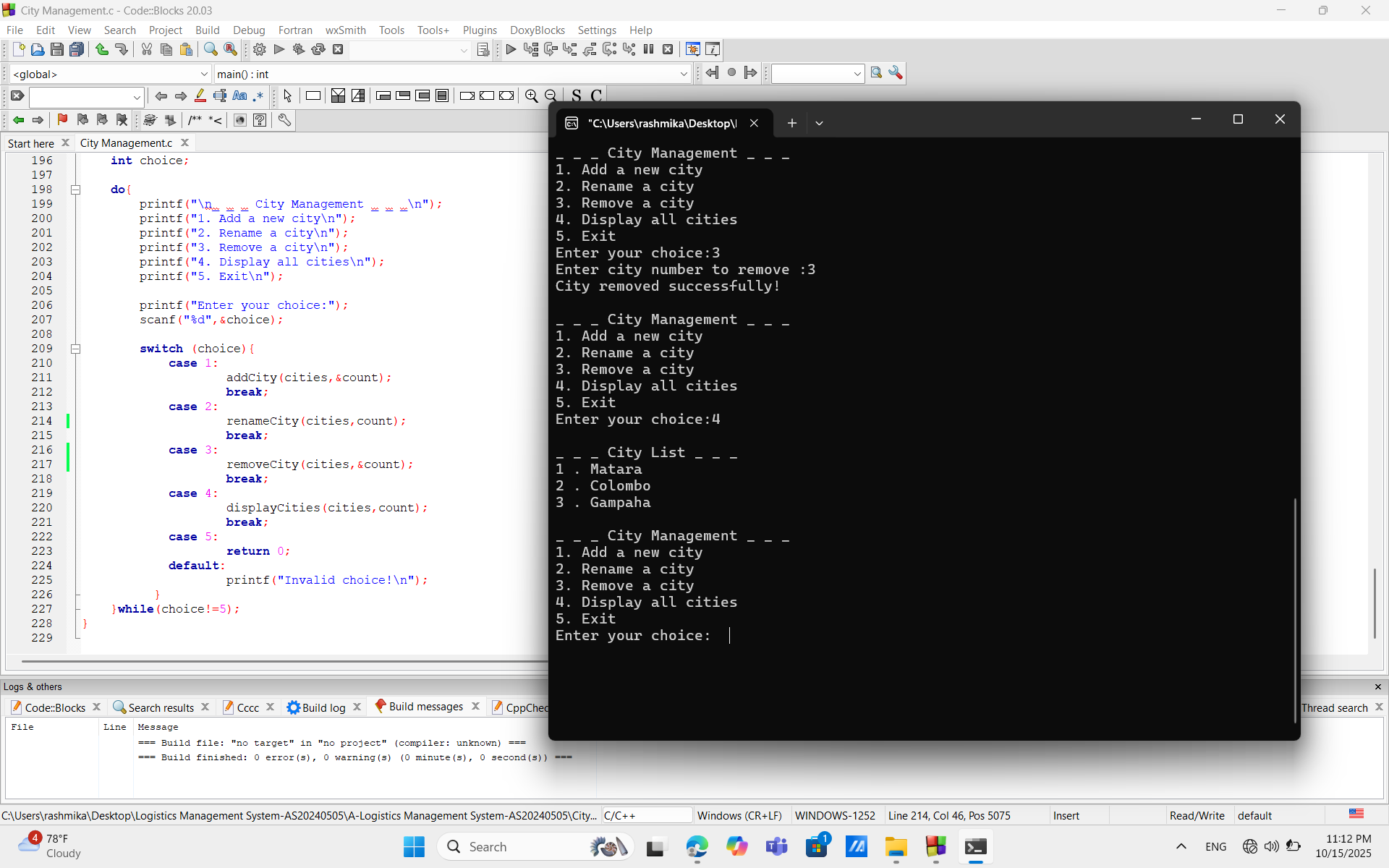
**Prototype:**

void renameCity(char cities[][50], int count);

**Parameters:**

* cities[][50] – A 2D character array containing the city names.
* count – The current number of cities in the list.

**How It Works:**

1. **Check if list is empty**
   * If count is 0, it displays: “No cities to rename.”
2. **Select city to rename**
   * Prompts the user to enter the city number.
   * If the number is invalid (less than 1 or greater than count), it displays: “Invalid city number.”
3. **Input new name**
   * Takes the new city name from the user into newName.
4. **Check uniqueness**
   * Compares newName with all existing cities.
   * If a duplicate is found, unique = 0.
5. **Rename city**
   * If the name is unique, copies newName into the selected city’s array (cities[cityNumber-1]).
   * Displays: “City renamed successfully!”
6. **Duplicate case**
   * If the new name already exists, displays: “City name already exists!”

### ****Function: removeCity****

**Purpose:**  
The purpose of this function is to **remove an existing city from the city list** and adjust the list accordingly.

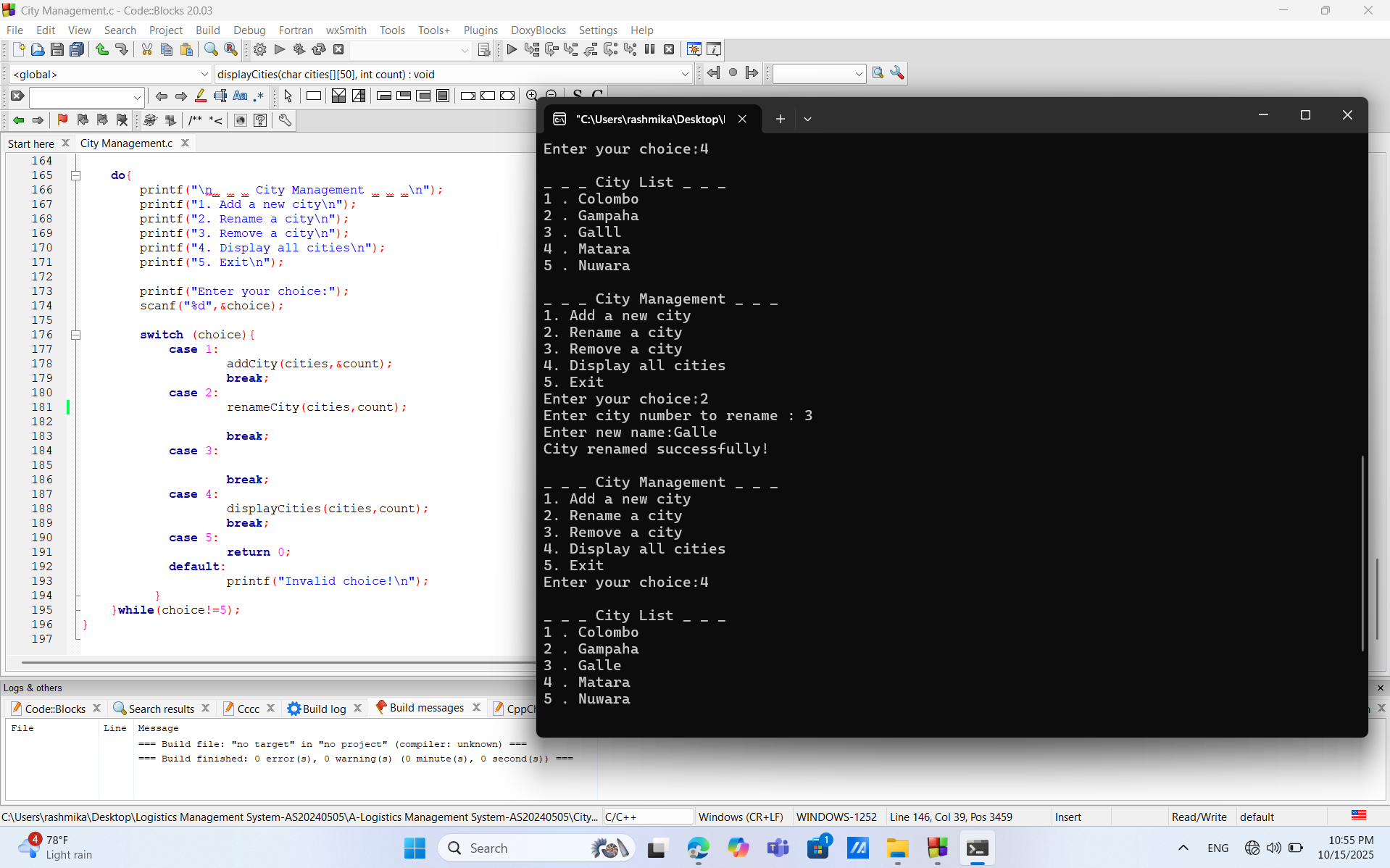
**Prototype:**

void removeCity(char cities[][50], int \*count);

**Parameters:**

* cities[][50] – A 2D character array containing the city names.
* \*count – A pointer to the current number of cities in the list.

**How It Works:**

1. **Check if list is empty**
   * If \*count is 0, it displays: “No cities to remove.”
2. **Select city to remove**
   * Prompts the user to enter the city number.
   * If the number is invalid (less than 1 or greater than \*count), it displays: “Invalid city number.”
3. **Remove city**
   * Shifts all cities after the selected city one position up in the array to fill the gap.
   * Decrements the city count ((\*count)--).
4. **Confirmation**
   * Displays: “City removed successfully!”

### ****Function: initializeDistances****

**Purpose:**  
The purpose of this function is to **initialize the distance matrix** used to store the distances between cities. It sets all distances to default values before any data is entered.

**Prototype:**

void initializeDistances(int distance[][30], int numCities);

**Parameters:**

* distance[][30] – A 2D integer array representing the distance between each pair of cities.
* numCities – The current number of cities in the system.

**How It Works:**

1. **Loop through all city pairs**
   * Uses two nested for loops to access every element in the distance matrix.
2. **Set default distances**
   * If both city indexes are the same (i == j), the distance is set to **0** (distance from a city to itself).
   * Otherwise, the distance is set to **-1**, meaning the distance between those two cities has **not been entered yet**.

**Purpose:**  
This function is used to **enter or update the distance** between two cities in the system. It allows the user to modify the distance matrix with correct values for city pairs.

**Prototype:**

void inputOrEditDistance(int distance[][30], int numCities);

**Parameters:**

* distance[][30] – A 2D array that stores the distance between every pair of cities.
* numCities – The total number of cities currently in the system.

**How It Works:**

1. **User Input:**
   * The function asks the user to enter two city numbers (from 1 to numCities).
   * It checks whether the entered city numbers are valid.
2. **Validation:**
   * If either city number is out of range → prints “Invalid city numbers.”
   * If both city numbers are the same → prints “Distance from a city to itself is always 0.”
3. **Update Distance:**
   * If the input is valid, it asks the user to enter the distance d between the two cities.
   * Updates the matrix in both directions:
   * distance[city1 - 1][city2 - 1] = d;
   * distance[city2 - 1][city1 - 1] = d;

This ensures the distance is **symmetric** (same both ways).

1. **Output Message:**
   * A computer screen with a black screen

     Description automatically generatedDisplays “Distance updated successfully!” when the entry is completed.

### ****Function: displayDistanceTable****

**Purpose:**  
The purpose of this function is to **display the distance matrix** between all cities in a tabular format, making it easy to view the distances at a glance.

**Prototype:**

void displayDistanceTable(int distance[][30], int numCities);

**Parameters:**

* distance[][30] – A 2D array containing distances between city pairs.
* numCities – The total number of cities currently in the system.

**How It Works:**

1. **Print table header:**
   * Prints the column headers (C1, C2, C3, …) representing each city.
2. **Print each row:**
   * Iterates through each city as the row index.
   * Prints the row header (C1, C2, …).
3. **Print distances:**
   * For each cell in the row:
     + If the value is -1 → prints ... to indicate the distance has not been entered.
     + Otherwise → prints the numeric distance with proper spacing.
4. **Output format:**
   * Produces a neat table with both row and column labels for all cities.

A computer screen shot of a black screen

Description automatically generated**Example Output:**

**Function: displayVehicles**

**Purpose:**  
This function displays a formatted list of all available vehicles along with their specifications, such as capacity, rate per kilometer, average speed, and fuel efficiency.

**Prototype:**

void displayVehicles(char vehicles[][20], int capacity[], int ratePerKm[], int avgSpeed[], int fuelEfficiency[], int size);

**Parameters:**

* vehicles[][20] – A 2D character array storing vehicle type names (e.g., Van, Truck, Lorry).
* capacity[] – An array holding the load capacity of each vehicle in kilograms.
* ratePerKm[] – An array storing the rate per kilometer in LKR.
* avgSpeed[] – An array representing the average speed of each vehicle.
* fuelEfficiency[] – An array for the fuel efficiency (in km/l).
* size – The total number of vehicle types available.

**How It Works:**

1. Prints the table header with column titles for all vehicle attributes.
2. Loops through each vehicle type and displays all details in a structured table format.
3. Allows users to easily compare vehicle performance and pricing.

**Example Output:**

Available Vehicles:

Type Capacity(kg) Rate/km(LKR) Speed(km/h) Efficiency(km/l)

Van 1000 30 60 12

Truck 5000 40 50 6

Lorry 10000 45 40 4

A computer screen shot of a black screen

Description automatically generated

**Function: calculateCost**

**Purpose:**  
This function calculates the total transportation cost based on the selected vehicle type and the travel distance.

**Prototype:**

float calculateCost(int ratePerKm[], int choice, float distance);

**Parameters:**

* ratePerKm[] – Array containing rate per kilometer for each vehicle.
* choice – The user’s selected vehicle number (1-based index).
* distance – The total distance to be traveled in kilometers.

**How It Works:**

1. Multiplies the distance by the rate per kilometer of the chosen vehicle.
2. Returns the total transportation cost as a floating-point value.

**Formula:**

Cost = Distance × Rate per km

**Example:**  
If a truck (rate = 40 LKR/km) is selected for a 50 km trip,

Cost = 50 × 40 = 2000 LKR

A computer screen shot of a black screen

Description automatically generated