**Logistics Management System**

**CSC 1012 Introduction to Computer Programming**

**A yellow and red circular logo

Description automatically generated**

**Project Title:** Logistics Management System (C language)

**Name:** M.I.Senuri Lakshani

**Date:** 2024.10.25

**GitHub Repository:**

1. **Introduction**

The **Logistics Management System** is a menu-driven software application developed as part of the CSC 1012 course at the University of Sri Jayewardenepura, Faculty of Applied Sciences. This system simulates basic logistics and delivery management operations. Using the C programming language, it manages cities, routes, vehicle types, delivery orders, costs, and performance reports.

The system is implemented using fundamental programming concepts such as arrays, loops, conditionals, and functions. Each function performs a specific task, ensuring the code is clear, reusable, and maintainable. This project demonstrates the practical application of programming concepts to solve real-world logistics problems, including cost estimation, route optimization, and performance tracking.

1. **Project Objective**

The main objective of this project is to design and implement a **menu-driven logistics management system** using the **C programming language**. This system is developed to demonstrate the practical application of fundamental programming concepts such as **arrays, functions, loops,** and **conditional statements** in solving real-world logistics problems.

The proposed system aims to simulate a simple yet effective **logistics and delivery management platform** that can manage cities, delivery routes, vehicle types, fuel consumption, and order costs efficiently.

Specifically, the program is designed to:

* Manage cities and the distances between them.
* Handle customer delivery requests by allowing users to select routes and vehicles.
* Estimate delivery time and cost based on distance, weight, and vehicle type.
* Track completed deliveries and generate performance reports for analysis.

Through this project, the goal is to build a foundational understanding of structured programming while applying it to a real-world logistics scenario, enhancing both problem-solving and system design skills.

1. System Design

Global Variables and Definitions

* Constant Definitions

The following symbolic constants are defined using the **#define** directive to set program limits and improve code readability and maintainability.

* **MAX\_CITIES (30):** Defines the maximum number of cities that can be managed by the system.
* **MAX\_DELIVERIES (50):** Specifies the maximum number of delivery records that can be stored.
* Global Variables

The program uses several global variables to store city details, delivery data, and distance information across multiple functions. This ensures that data can be accessed and updated throughout the program without repeatedly passing large arrays as parameters.

* Delivery Record Arrays

These global arrays are used to maintain complete information about each delivery, such as the origin, destination, vehicle type, cost, and time details.

* Vehicle Data

Global arrays are defined to represent vehicle types and their corresponding specifications.  
These are used across functions to calculate delivery cost, speed, and fuel efficiency.

1. Modules Description & Sample Outputs

Main Program Structure and Menu

The program is menu-driven with the following top-level menu:

1. City Management
2. Distance Management
3. Vehicle Management
4. Delivery Request Handling
5. Least-Cost Route Finder
6. Performance Reports
7. Exit

Each menu item calls a set of modular functions. Modular design improves readability and maintainability.

1. Main Menu

**Purpose:**  
This module provides the user with access to the main functionalities of the program. Through the menu options, the user can perform the following operations:

* City management
* Distance management
* Vehicle management
* Delivery request handling
* Least-cost route finding
* Performance reports
* Program exit

**Input:**  
The user enters an integer value representing their choice. Valid inputs are in the range of 1–7.

**Process :**

1. Initialize the numCities variable when the program starts.
2. Repeat the menu using a do-while loop until the user selects the exit option (7).
3. Use a switch-case statement to handle the user’s choice.
4. Call the corresponding function for each valid choice:
   * cityManagement()
   * distanceManagement()
   * vehicleManagement()
   * handleDeliveryRequest()
   * findLeastCostRoute()
   * performanceReports()
5. Terminate the loop when the exit option (7) is selected.
6. Display an error message for invalid input in the default case.

A computer screen shot of a black screen

Description automatically generated

1. City Management system

* **addCity()**

**Purpose:**  
To add a new city to the city list, ensuring no duplicates and the list does not exceed the maximum allowed cities.

**Inputs:**

* cities[][50] → 2D array storing names of all cities.
* \*numCities → Pointer to the current number of cities.

**Process:**

1. Check if the city list is full (\*numCities >= MAX\_CITIES). If yes, display an error.
2. Take input for the new city name.
3. Check if the city name already exists by comparing with each existing city.
4. If the name is unique, copy it to the cities array at the next available index.
5. Increment \*numCities and display success message; otherwise, display "City name already exists".

A screenshot of a computer

Description automatically generated

* **displayCities()**

**Purpose:**  
To display all the cities currently in the list with their serial numbers.

**Inputs:**

* cities[][50] → 2D array of city names.
* numCities → Total number of cities.

**Process:**

1. Check if there are any cities to display.
2. Loop through the cities array and print each city with its index number.

* **cityManagement()**

**Purpose:**  
To provide a user interface for managing city operations: adding, renaming, removing, and displaying cities.

**Inputs:**

* None directly (uses global cities array and numCities variable).

**Process:**

1. Display a menu with options: Add, Rename, Remove, Display, Exit.
2. Take user choice as input.
3. Call the corresponding function based on the user's choice.
4. Repeat until the user chooses to exit (option 5).
5. Display error for invalid choices.

A computer screen with a black screen

Description automatically generated

* **renameCity()**

**Purpose:**  
To rename an existing city while ensuring the new name is unique.

**Inputs:**

* cities[][50] → 2D array of city names.
* numCities → Total number of cities.

**Process:**

1. Check if there are any cities to rename.
2. Take input for the city number to rename and validate it.
3. Take input for the new name.
4. Check if the new name already exists among other cities.
5. A screenshot of a computer

   Description automatically generatedIf unique, update the city's name in the array and display success; otherwise, display "City name already exists".

* **removeCity()**

**Purpose:**  
To remove a city from the list and shift the remaining cities up to maintain array order.

**Inputs:**

* cities[][50] → 2D array of city names.
* \*numCities → Pointer to the current number of cities.

**Process:**

1. Check if there are any cities to remove.
2. Take input for the city number to remove and validate it.
3. Shift all cities after the removed city one position up in the array.
4. Decrement \*numCities and display success message.

A screenshot of a computer

Description automatically generated

* **cityManagement()**

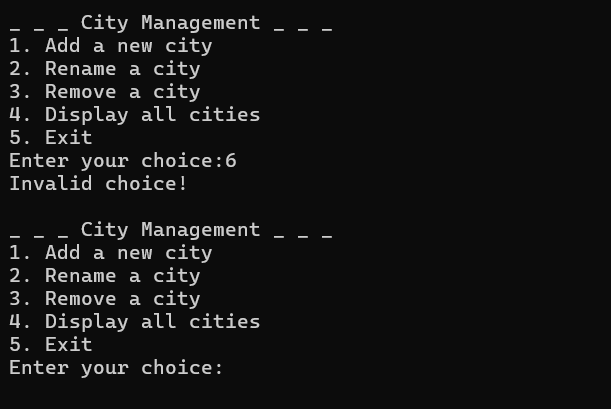
**Purpose:**  
To provide a user interface for managing city operations: adding, renaming, removing, and displaying cities.

**Inputs:**

* None directly (uses global cities array and numCities variable).

**Process:**

1. Display a menu with options: Add, Rename, Remove, Display, Exit.
2. Take user choice as input.
3. Call the corresponding function based on the user's choice.
4. Repeat until the user chooses to exit (option 5).
5. Display error for invalid choices.



1. **Distance Management System**

* initializeDistances()

**Purpose:**  
To initialize the distance table between cities.  
Distance from a city to itself is set to **0**, and all other distances are set to **-1** (indicating no distance entered yet).

**Inputs:**

* distance[][MAX\_CITIES] → 2D array storing distances between cities.
* numCities → Number of cities in the system.

**Process:**

1. Loop through all rows and columns of the distance array.
2. If i == j (same city), set distance to 0.
3. Otherwise, set distance to -1.

* **inputOrEditDistance()**

**Purpose:**  
To input or edit the distance between two cities.  
It updates the distance table symmetrically.

**Inputs:**

* distance[][MAX\_CITIES] → Distance table array.
* numCities → Number of cities in the system.

**Process:**

1. Ask the user for two city numbers (from 1 to numCities).
2. Validate the city numbers.
3. If the same city is selected, display “Distance from a city to itself is always 0”.
4. Otherwise, ask for the distance between the two cities.
5. Update the distance table for both directions (distance[city1-1][city2-1] and distance[city2-1][city1-1]).
6. Display “Distance updated successfully!”.

* displayDistanceTable()

**Purpose:**  
To display the distance table showing all city-to-city distances.

**Inputs:**

* distance[][MAX\_CITIES] → Distance table array.
* numCities → Number of cities in the system.

**Process:**

1. Print the header “\_ \_ \_ Distance Table \_ \_ \_”.
2. Print the top row with city numbers (C1, C2, C3…).
3. Loop through the distance array:
   * If the distance is -1, display “…”.
   * Otherwise, display the actual distance.
4. Print each row for a city with distances to all other cities.

* DistanceManagement()

**Purpose:**  
To provide a user interface for managing distances between cities.

**Inputs:**

* None directly (uses global distanceTable and numCities).

**Process:**

1. Initialize the distance table using initializeDistances().
2. Display a menu:
   * 1 → Input or Edit Distance
   * 2 → Display Distance Table
   * 3 → Exit
3. Take user choice as input.
4. Call the corresponding function (inputOrEditDistance() or displayDistanceTable()).
5. Repeat the menu until the user chooses to exit.
6. Show “Invalid choice!” for invalid input.

A screenshot of a computer

Description automatically generated

1. Vehicle Management System

* **displayVehicles()**

**Purpose:**  
To display a list of available vehicles along with their specifications such as capacity, rate per km, average speed, and fuel efficiency.

**Inputs:**

* Uses global arrays: vehicles[], capacity[], ratePerKm[], avgSpeed[], fuelEfficiency[].
* No direct input from the user.

**Process:**

1. Print a header: “Available Vehicles” with column titles.
2. Loop through all vehicles (3 in this example).
3. For each vehicle, print the type, capacity, rate per km, average speed, and fuel efficiency.

* calculateCost()

**Purpose:**  
To calculate the delivery cost for a selected vehicle based on distance.

**Inputs:**

* choice → Vehicle number selected by the user (1-3).
* distance → Distance for the delivery in km.

**Process:**

1. Multiply the vehicle’s rate per km (ratePerKm[choice-1]) by the distance.
2. Return the calculated cost.

* vehicleManagement()

**Purpose:**  
To manage vehicle selection and cost calculation for deliveries.

**Inputs:**

* User choice of vehicle (1-3).
* Delivery distance in km.

**Process:**

1. Display the list of available vehicles using displayVehicles().
2. Ask the user to select a vehicle number. Validate the choice (must be 1-3).
3. Ask the user to enter the distance in km.
4. Call calculateCost() to compute the cost.
5. Display the selected vehicle, distance, and calculated cost.
6. Show “Invalid choice.” if the vehicle number is out of range.
7. A screen shot of a computer

   Description automatically generatedDelivery Request Handling

* handleDeliveryRequest()

**Purpose:**  
To manage a delivery request by calculating costs, fuel consumption, estimated delivery time, and final customer charge based on the selected cities, vehicle, and weight.  
Also records the delivery details for future reference.

**Inputs:**

* User input:
  + Source city index
  + Destination city index
  + Vehicle type (1=Van, 2=Truck, 3=Lorry)
  + Delivery weight (kg)
* Uses global variables:
  + cities[], distanceTable[][], vehicles[], capacity[], ratePerKm[], avgSpeed[], fuelEfficiency[]
* Maximum deliveries: MAX\_DELIVERIES

**Process:**

1. Check if at least 2 cities exist; if not, display error.
2. Display available cities and take user input for source and destination cities. Validate input and check that distance is set.
3. Display available vehicles and take user input for vehicle type. Validate choice.
4. Take delivery weight input and ensure it does not exceed the selected vehicle’s capacity.
5. Calculate delivery parameters:
   * **Delivery Cost** = Distance \* RatePerKm \* (1 + Weight/10000)
   * **Estimated Time (hours)** = Distance / AvgSpeed
   * **Fuel Used (L)** = Distance / FuelEfficiency
   * **Fuel Cost** = FuelUsed \* FuelPrice
   * **Total Operational Cost** = DeliveryCost + FuelCost
   * **Profit** = TotalOperationalCost \* 0.25
   * **Customer Charge** = TotalOperationalCost + Profit
6. Display detailed delivery estimation including all calculated costs, profit, vehicle, distance, and estimated time.
7. Record delivery details into global arrays (deliveryFrom[], deliveryTo[], deliveryVehicle[], deliveryWeight[], deliveryDistance[], deliveryCostRecord[], deliveryFuelCost[], deliveryTotalCost[], deliveryProfit[], deliveryCustomerCharge[], deliveryTime[]).
8. Increment deliveryCount to track the total number of deliveries.

* deliveryRequestHandling()

**Purpose:**  
To provide a simplified interface for handling delivery requests.

**Inputs:**

* None directly; it calls handleDeliveryRequest(), which handles all inputs.

**Process:**

1. Call handleDeliveryRequest() to manage the delivery request process.

A screenshot of a computer

Description automatically generated

1. Finding The Least-Cost Route (Least-Distance)

* findLeastCostRoute()

**Purpose:**  
To find the least-cost route (shortest distance) between a source city and a destination city, optionally passing through intermediate cities.  
Currently supports a maximum of 4 cities for simplicity.

**Inputs:**

* User input:
  + Source city index
  + Destination city index
* Uses global variables:
  + cities[], distanceTable[][], numCities

**Process:**

1. Check if there are at least 2 cities.
2. Check if the number of cities is ≤ 4; otherwise, show a message that only up to 4 cities are supported.
3. Display available cities.
4. Take user input for source and destination cities and validate.
5. Identify intermediate cities (all cities except source and destination).
6. Generate all permutations of the intermediate cities to find every possible path from source → intermediate(s) → destination.
7. For each permutation, calculate the total distance using totalDistance().
8. Track the permutation with the smallest total distance (bestDist).
9. Display the least-cost route path and total distance.

* totalDistance()

**Purpose:**  
To calculate the total distance of a given path (sequence of cities).

**Inputs:**

* path[] → Array of city indices representing a path.
* len → Number of cities in the path.

**Process:**

1. Initialize dist = 0.
2. Loop through each pair of consecutive cities in the path.
3. If a distance is -1 (not set), return -1 (invalid path).
4. Otherwise, add the distance to the total.
5. Return the total distance of the path.

* permute()

**Purpose:**  
To generate all permutations of intermediate cities and determine the path with the least total distance.

**Inputs:**

* arr[] → Array of intermediate city indices.
* l → Left index for permutation recursion.
* r → Right index for permutation recursion.
* source → Source city index.
* destination → Destination city index.
* bestDist → Pointer to the best distance found so far.
* bestPath[] → Array to store the best path.
* intermediateCount → Number of intermediate cities.

**Process:**

1. If l == r, form a complete path by combining source → permutation of intermediate cities → destination.
2. Calculate total distance using totalDistance().
3. If distance is valid and less than bestDist, update bestDist and store the path in bestPath[].
4. Otherwise, recursively swap elements to generate all permutations.
5. A computer screen with white text

   Description automatically generatedContinue recursion until all permutations are tested.
6. Performance Reports

* performanceReports()

**Purpose:**  
To generate a performance summary of all completed deliveries, including total deliveries, total distance, average delivery time, revenue, profit, and information about the longest and shortest routes.

**Inputs:**

* Uses global arrays:
  + deliveryDistance[], deliveryTime[], deliveryCustomerCharge[], deliveryProfit[], deliveryFrom[], deliveryTo[]
* deliveryCount → Number of completed deliveries.

**Process:**

1. Check if any deliveries have been completed (deliveryCount == 0). If none, display “No deliveries completed yet” and exit.
2. Initialize cumulative variables: totalDistance, totalTime, totalRevenue, totalProfit.
3. Initialize longest and shortest indices to track the longest and shortest routes.
4. Loop through all completed deliveries:
   * Add each delivery’s distance, time, revenue, and profit to the totals.
   * Update longest if the current delivery distance is greater than the current longest.
   * Update shortest if the current delivery distance is less than the current shortest.
5. Calculate average delivery time (totalTime / deliveryCount).
6. Display a formatted performance report including:
   * Total deliveries completed
   * Total distance covered
   * Average delivery time
   * Total revenue
   * Total profit
   * Longest route completed with cities and distance
   * A computer screen with white text

     Description automatically generatedShortest route completed with cities and distance

Conclusion

The **Logistics Management System** successfully demonstrates the implementation of a basic logistics and delivery management application using the C programming language. The system allows efficient management of cities, distances, vehicles, and delivery orders, while also providing accurate cost, fuel, and time estimations.

Through this project, key programming concepts such as arrays, loops, conditionals, and modular functions were applied in a practical scenario. The system also tracks delivery performance, generates reports, and identifies least-cost routes, showcasing problem-solving and analytical skills.

Overall, this project provides a functional and modular software solution for small-scale logistics operations, highlighting the importance of systematic programming and data management in real-world applications.

References

1. University of Sri Jayewardenepura, Faculty of Applied Sciences. (2025). *CSC 1012 – Introduction to Computer Programming Course Material*.
2. GeeksforGeeks. (n.d.). *C Programming Language Tutorials*. Retrieved from https://www.geeksforgeeks.org/c-programming-language/
3. TutorialsPoint. (n.d.). *C Programming Tutorial*. Retrieved from https://www.tutorialspoint.com/cprogramming/index.htm
4. GitHub Repository of Project (Your project repository URL here).