**DSLT ASSIGNMENT: Robotic Line simulator**

K K Bharadwaj

ME24B1023

13th April, 2025

**Objectives:**

· To create a highly effective data management program to handle a considerable number of tasks, including edge cases in the realm of automotive production.

· To use several different implementations of data structures for each specific subsystem, including **Queues** *(for Part delivery system, queuing each part for processing)*, **Stacks** *(Used as a LIFO parts picker for assembly)*, **Arrays** *(Storage units for completed prototypes, shipping out the oldest version once storage limit is reached),* **Linked Lists** *(Singly LL used for defective protypes, Doubly LL for repaired, Circular LL for VIPs)*

**Design explanation:**

To explain the different design choices takes in the code, I shall divide this section into 2 parts i.e Why specific data structures and how they solve the problems efficiently

· The different data structures used and reasoning: As given in the question,

1) We first use Queues for the part delivery system. This is advantageous as we may simply “Funnel” parts as they are received from user input, with each part being presented in the precise order they were received in (i.e FIFO).

2) For the assembly line task management, we use Stack implementation. We use stacks simply due to ease of use, as we may pop elements (here, parts) as required for assembly

3) For storage of units, we use Arrays. Arrays are not dynamic memory types and thus we may set the maximum storage that may be used for protype car storage

4) Finally, we use Linked lists, within which we use Singly linked list for defective protypes, doubly linked lists for repaired protypes and for VIPs, we use circular linked list as they must be given precedence.

· How efficiency is achieved:

To understand how this allows for efficient function, we must understand the role of each data structure used in its implementation

1) Queue: As stated before, allows us to funnel parts in without significant changes

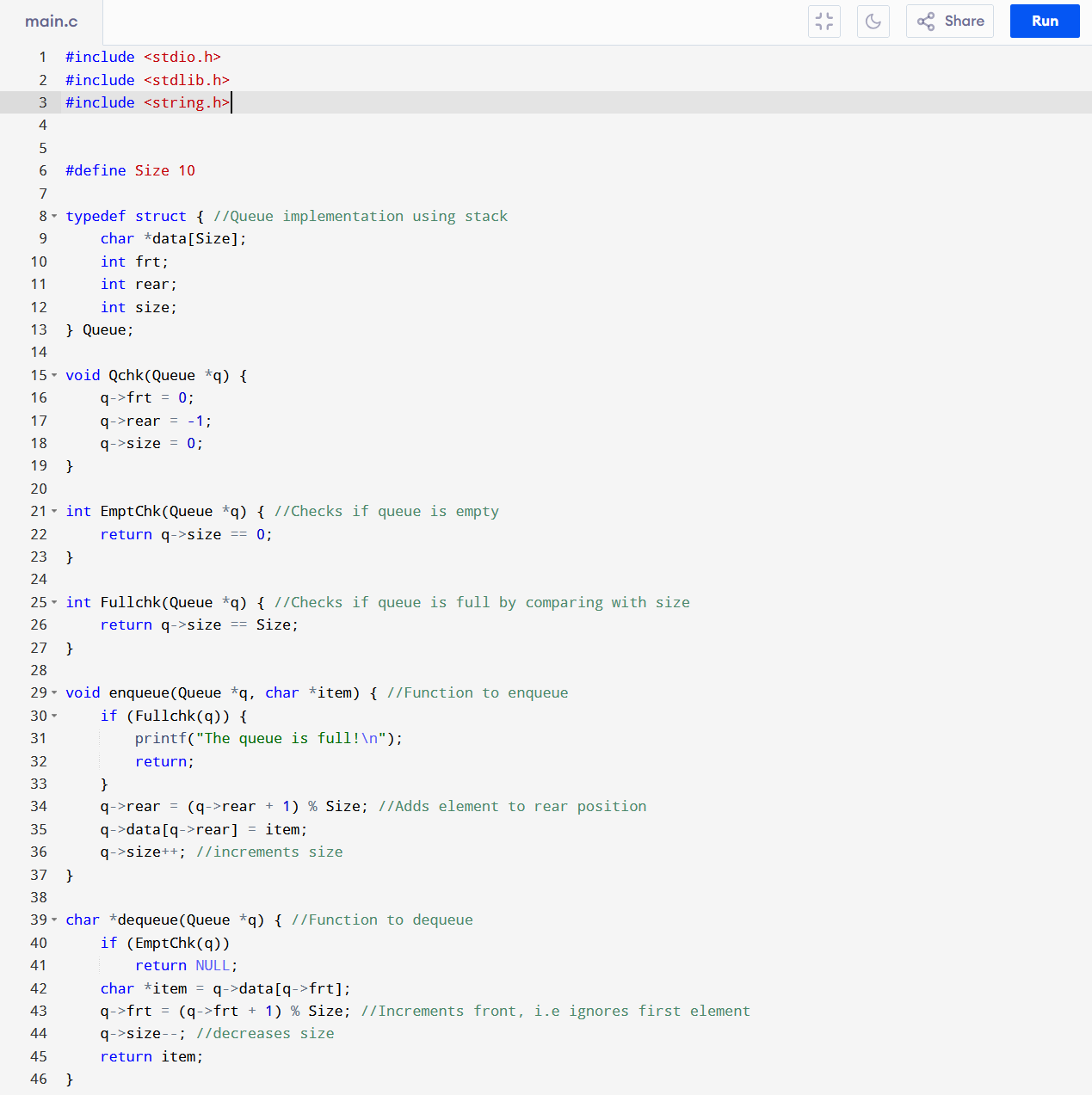
2) Stacks: Allows robot to simply take element of queue and place it within stacks. Immensely simplified data management as elements can be orderly accessed based on need

3) Arrays allow us to fix a size, allowing us to simply push out prototypes and based on requirement remove the oldest ones upon reaching the storage cap.

4) The several different types of linked list offer us advantages, not just limited to reducing time complexity, reducing redundant memory usage, allowing us to add priority to different functions and so on.

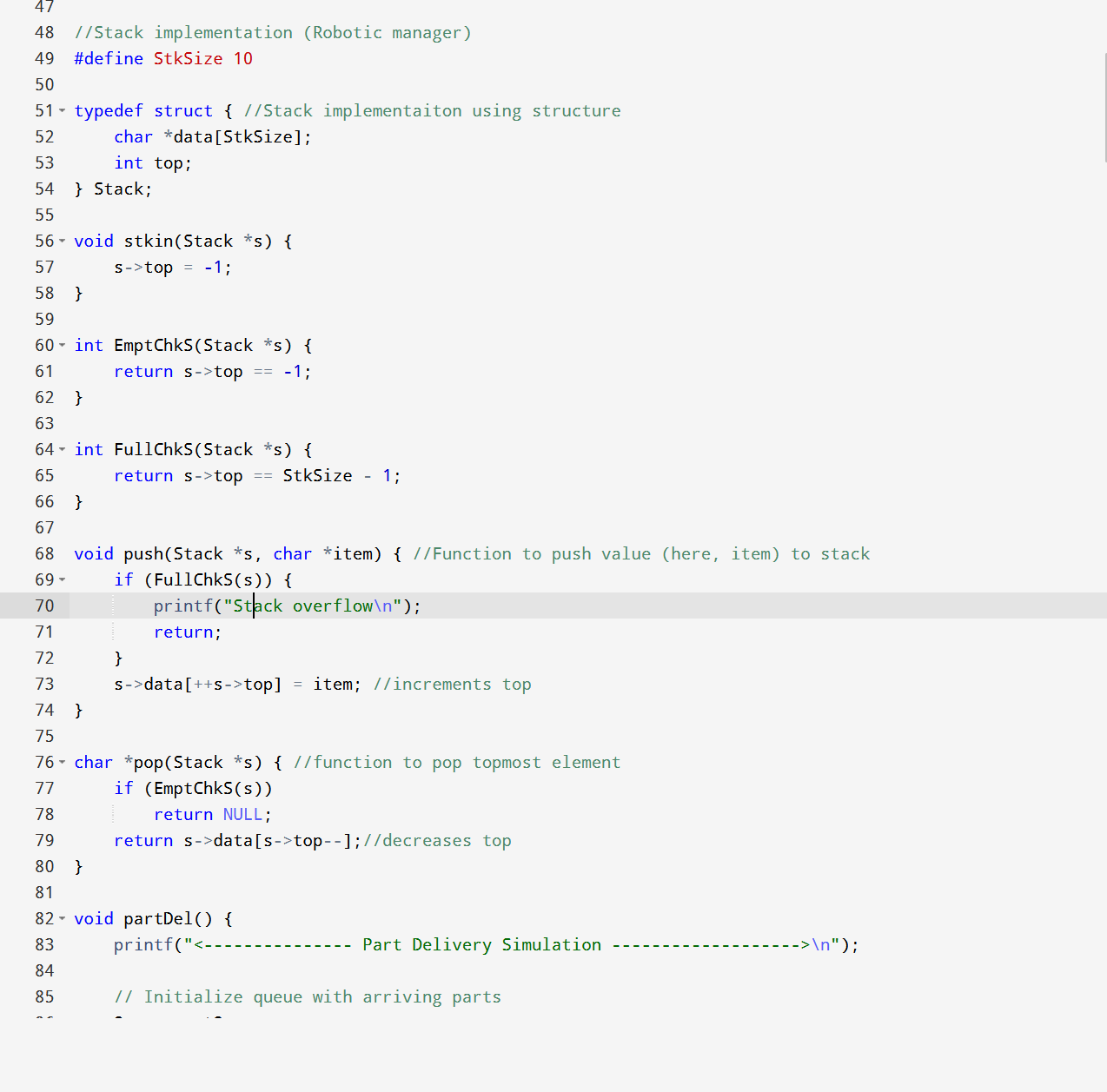
**Code Logic:**

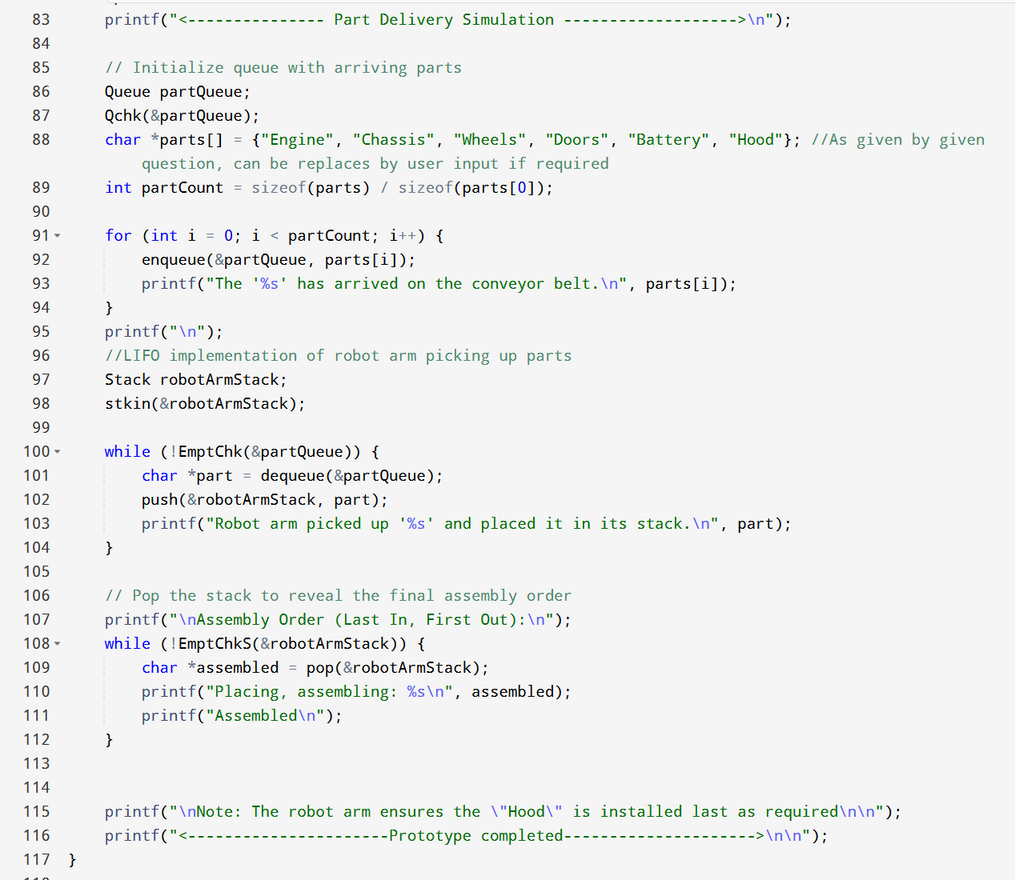
The code logic shall be explained with the aid of screen shots.



First, we observe standard queue implementation, i.e we have Enqueue, Dequeue as checks to see if the queue is full or empty. Given user input, I have designed queue around it, however it is capable of handling any user input.

Next, we observe stack implementation. Here, all is normal, however,





We observe that for Part simulation, the code follows the following logic

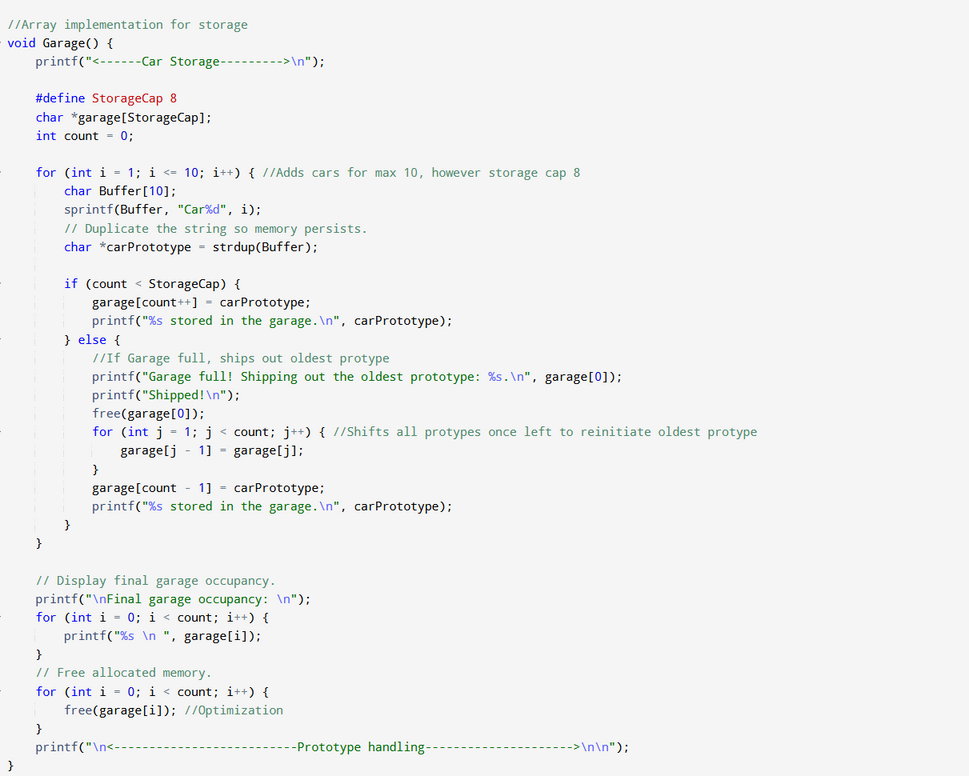
→Initialize Queue, user input added to queue element by element

→ Acknowledge addition of elements to queue

→ Use LIFO Implementation (stack) to simulate robot picking parts of the conveyor belt and placing in stack

→ The picking up of parts involves dequeuing of elements and pushing to stack

->Elements are then popped from stack to show storage order



Here, array implementation follows the following steps:

→ Prototypes completed of the line are designated as Car 1, Car 2, Car 3 and so on

→ Here, Garage is our array, elements are added to array from carPrototype

->iterates through using count++

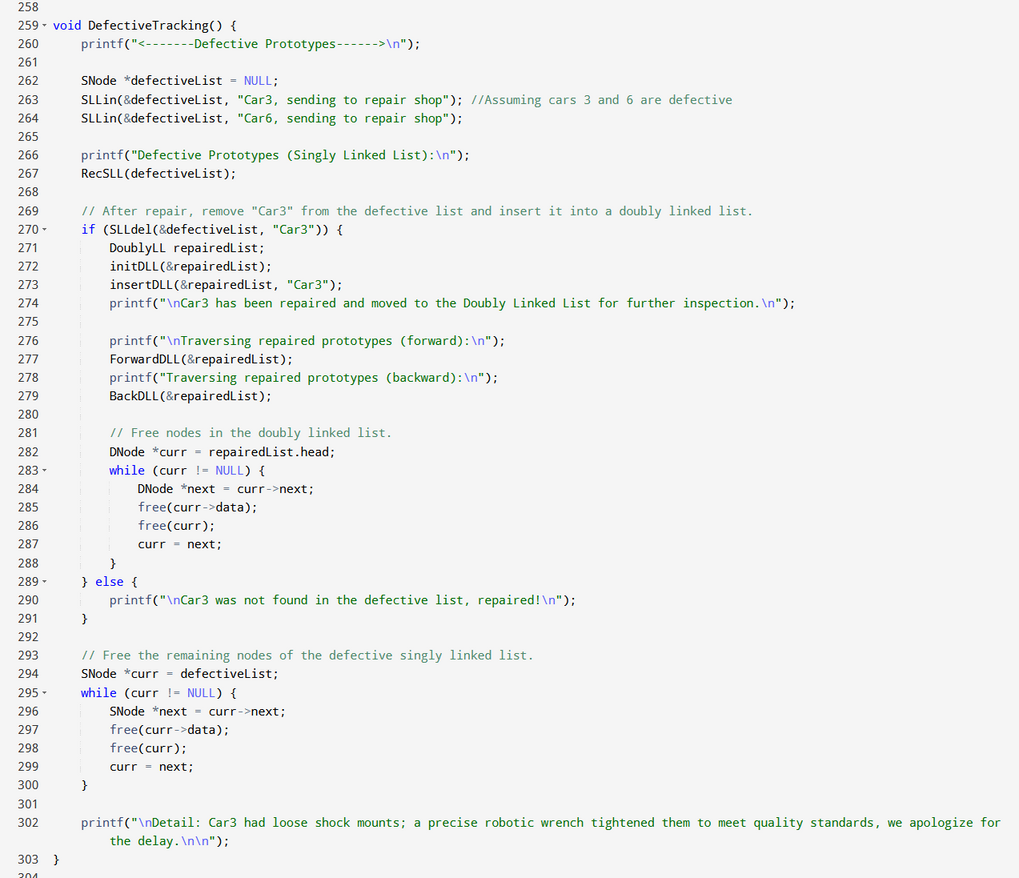
->Once full, ships out first element (which will be oldest prototype)

→ Shifts all elements back to make previous second element as current oldest prototype

->Displays final garage occupancy by iterating through array and printing it’s elements



Singly linked list implementation is standard, where each node refers to a defunct prototype that is categorized.



Standard doubly linked list implementation

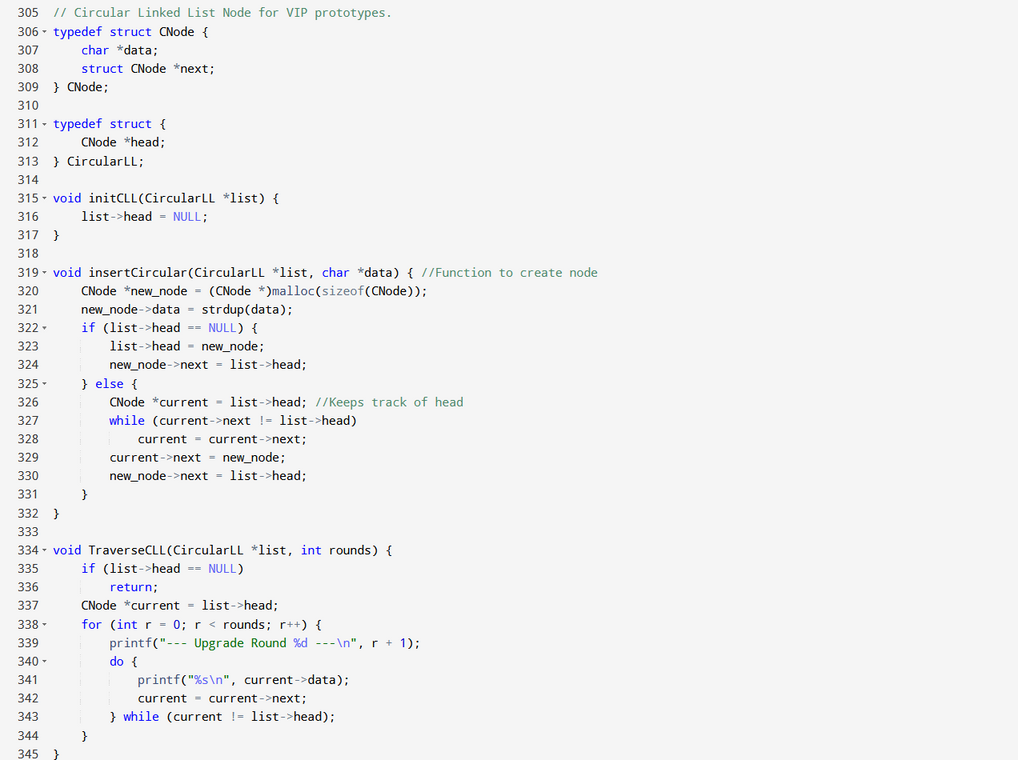
Here, the function defective tracking follows the following logic:

→ Assuming cars 3 and 6 are defective, defective-list (Here an SLL) is used to categorize and store

→ Once stored, we pass them to Doubly linked list (Repaired list) to then display as repaired (consider lines 270 to 274)

→ We traverse through the prototypes present within the DLL using Forward DLL and backward DLL.

→ To optimize memory managment, we use free to then delete the remaining cars within as they are no longer needed within (Case in point, line 290, showing car3 after repairs has been shipped out)



Standard CLL implementation ,however we use traverse CLL in only one way. We also note that we include a count (upgrade rounds) for each exclusive car

Within memory CLL, we perform two actions, once perform linking and freeing memory

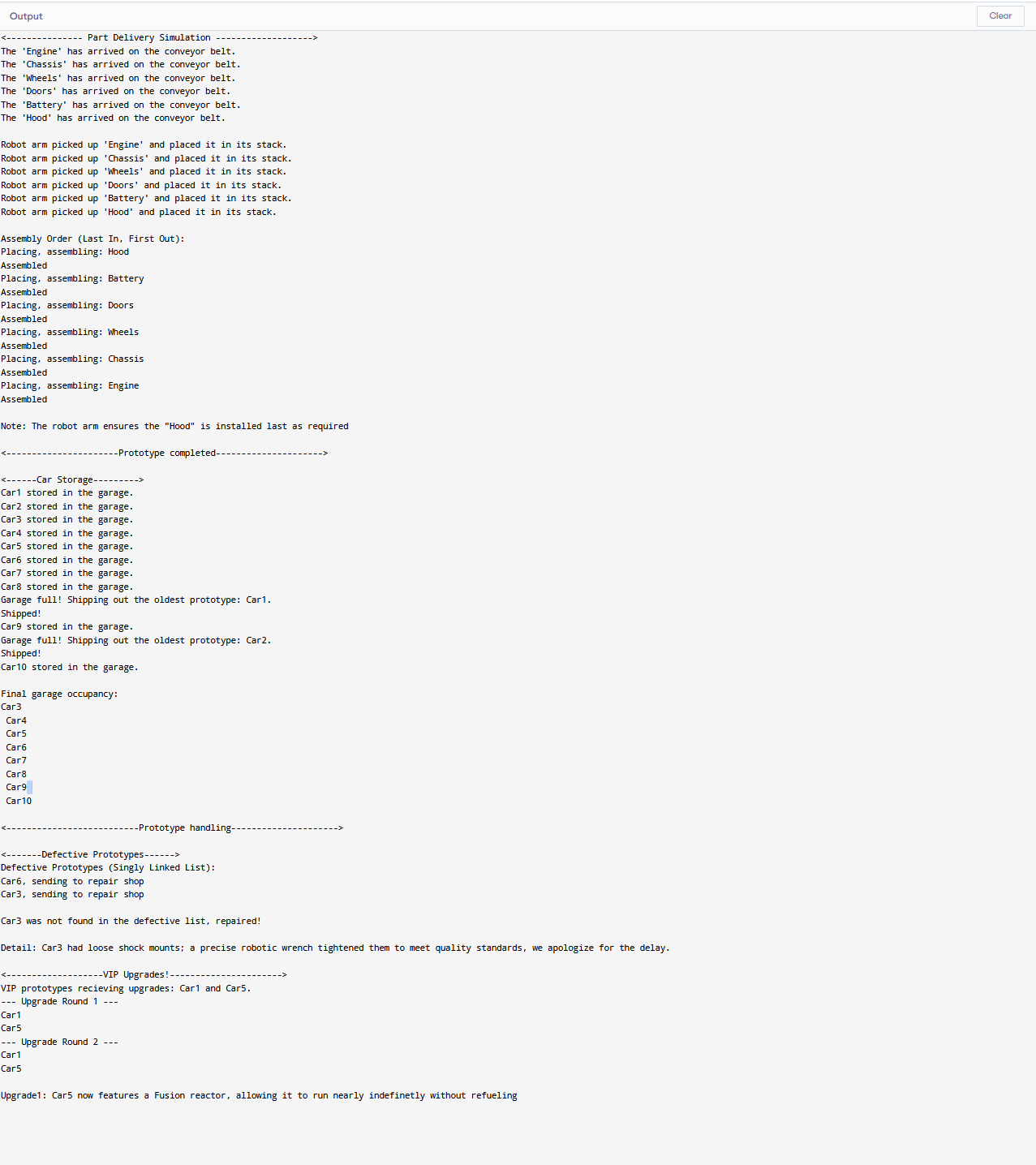
Logic followed in VIPS is as follows:

→ We obtain user input cars to perform upgrades for, then add to Viplist (CLL)

→ After storage and “alteration”, we remove the cars use MemCLL to free memory and print the “upgrades” that we had done to the cars

This is the logic of the overall code

Sample output:

Fin