ParkingLot System - Technical API Documentation

`parkinglot.cpp`

The logic core of the project, containing:

`Car` Class

The `Car` class serves as the fundamental data unit and node structure within the parking lot simulation system. Each instance represents a single vehicle that can be stored in either queue or stack data structures, containing identification information and linkage capabilities for forming connected lists. The class provides minimal encapsulation with public members for direct access, focusing on efficiency over strict data hiding in this simulation context.

`id` (Integer)

This public integer member stores the unique numerical identifier for each vehicle, serving as the primary key for all system operations including insertion, retrieval, and location tracking. Valid vehicles use positive integers while the value -1 indicates an invalid or placeholder state. The identifier enables efficient vehicle lookup and duplicate prevention throughout the parking system's queue and stack structures.

`model` (String)

This public string member contains descriptive information about the vehicle's make or type, providing human-readable identification alongside the numerical ID. While less critical for core operations than the ID field, it offers contextual vehicle information that could be displayed in user interfaces or system reports. The string utilizes standard library memory management without requiring manual resource handling.

`next` (Car Pointer)

This public pointer member enables the formation of linked data structures by referencing the subsequent Car object in a chain. In stack implementations, it creates downward links from top to bottom elements, while in queues it establishes forward sequencing from front to rear. The pointer is initialized to nullptr to indicate terminal nodes and must be properly managed during structural operations.

Default Constructor

The default constructor initializes a Car object with neutral, invalid values to represent an empty or uninitialized vehicle node. It sets the id to -1, model to an empty string, and next pointer to nullptr, ensuring default-constructed objects don't contain garbage values. This provides a safe baseline state for placeholder vehicles returned by failed operations.

Parameterized Constructor

This constructor creates a fully specified Car object with provided identification data and proper linkage defaults. It accepts an integer identifier and model description, directly assigning them to public members while setting the next pointer to nullptr. This is invoked when new vehicles enter the system, ensuring they begin as isolated nodes ready for insertion.

`MyStack` Class

The MyStack class implements a Last-In-First-Out data structure using a singly linked list of Car objects, modeling individual parking spots within the system. It manages bounded collections where only the topmost vehicle is immediately accessible, simulating real-world parking constraints with capacity-limited operations. The class includes sorting capabilities and proper memory management.

`size` (Integer)

This public integer tracks the current number of Car objects stored within the stack, incremented during pushes and decremented during pops. It enables constant-time evaluation of emptiness and fullness conditions by comparing against 0 and capacity respectively, providing immediate occupancy status without traversal.

`capacity` (Integer)

This public integer defines the maximum number of vehicles the stack can contain, representing the physical space limitation of a parking spot. Set during initialization and remaining constant, it establishes boundary conditions for push operations and determines when the stack must reject additional vehicles.

`top` (Car Pointer)

This public pointer references the most recently added Car object, serving as the head of the singly linked list implementing the stack structure. During push operations it updates to new cars with proper linkage adjustments, while pop operations move it to the next element. It's critical for traversal and cleanup operations.

Constructor

The constructor initializes an empty stack with size and capacity set to 0 and top pointer set to nullptr via initializer list. This creates a valid stack object ready for configuration with specific capacity limits before accepting elements, establishing proper empty state invariants.

`merge_sort` Method

This private recursive method implements the merge sort algorithm on a vector of Car objects within specified index bounds. It follows divide-and-conquer methodology: recursively splitting ranges until single elements, then merging sorted subarrays in ascending order by car ID. The algorithm provides O(n log n) complexity for sorting operations.

`isEmpty` Method

This method returns a boolean indicating whether the stack contains zero elements by comparing the size member against 0. It executes in constant time using the maintained counter rather than traversing the linked structure, preventing invalid pop operations on empty stacks.

`isFull` Method

This method returns a boolean indicating whether the stack has reached maximum capacity by comparing size against capacity. The constant-time check leverages the maintained size counter to determine if push operations should be rejected, maintaining data structure integrity.

`push` Method

This method inserts a new Car object at the top of the stack with capacity validation. It checks if the stack is full, allocates a new Car with provided parameters, sets its next pointer to current top, updates the top reference, increments size, and returns success status. It handles both allocation and linkage in one operation.

`pop` Method

This method removes and returns the topmost Car object if available, implementing LIFO removal semantics. It validates the stack isn't empty, stores the top pointer, creates a data copy for return, advances top to the next element, deletes the original node, decrements size, and returns the car copy by value.

`sort` Method

This public method orchestrates complete sorting of the stack's cars in ascending order by ID using merge sort. It dismantles the stack into a vector via repeated pops, calls merge_sort on the vector, then reconstructs the stack by pushing cars from the sorted vector's back. This achieves sorted order from top to bottom.

`print` Method

This method generates a formatted string representation of the stack's contents for debugging purposes. It traverses the linked list from top to bottom using an iterator pointer, appending each car's ID (formatted with width 2) followed by a separator to a string stream, returning the complete representation.

Destructor

The destructor implements comprehensive cleanup of all dynamically allocated Car objects to prevent memory leaks. It iterates through the linked list using the top pointer, storing each node in a temporary variable, advancing the pointer, and deleting the node. This linear process ensures complete deallocation.

`MyQueue` Class

The MyQueue class implements a First-In-First-Out data structure using a singly linked list of Car objects, modeling the waiting line before parking entry. It manages bounded sequences where new arrivals join at the rear and departures occur from the front, simulating fair queue ordering with capacity limitations and proper memory management.

`size` (Integer)

This public integer maintains the current number of Car objects in the queue, incremented during enqueue and decremented during dequeue operations. It enables constant-time evaluation of emptiness and fullness conditions and must be synchronized with pointer operations to accurately reflect contents.

`capacity` (Integer)

This public integer defines the maximum allowable vehicles in the queue, representing the physical constraint of the waiting area. Set during initialization and remaining constant, it establishes the boundary for enqueue operations and prevents queue overflow beyond simulated space limits.

`front` (Car Pointer)

This public pointer references the first Car object in the queue, representing the longest-waiting vehicle that will depart next. It serves as the access point for dequeue operations and must be properly managed during additions and removals, starting as nullptr for empty queues.

`rear` (Car Pointer)

This public pointer references the last Car object in the queue, representing the most recently arrived vehicle. It enables efficient O(1) enqueue operations by providing direct access to the list's end without traversal, and must be maintained consistently with the linked structure.

Constructor

The constructor initializes an empty queue with size and capacity set to 0 and both front and rear pointers set to nullptr via initializer list. This creates a functional queue object requiring separate capacity configuration before enforcing size limits during enqueue operations.

`isEmpty` Method

This method returns a boolean indicating whether the queue contains zero elements by comparing size against 0. The constant-time check uses the maintained counter to prevent invalid dequeue operations on empty queues, ensuring data structure integrity.

`isFull` Method

This method returns a boolean indicating whether the queue has reached maximum capacity by comparing size against capacity. This quick check determines if enqueue operations should be rejected due to space constraints, maintaining bounded queue behavior.

`enqueue` Method

This method inserts a new Car object at the rear of the queue with capacity validation. It checks if the queue is full, allocates a new Car with provided parameters, sets its next pointer to nullptr, updates front/rear pointers appropriately, increments size, and returns success status.

`dequeue` Method

This method removes and returns the frontmost Car object if available, implementing FIFO removal semantics. It validates the queue isn't empty, stores the front pointer, creates a data copy for return, advances front to the next element, handles empty queue transition, deletes the original node, decrements size, and returns the copy.

`print` Method

This method generates a formatted string representation of the queue's contents for debugging purposes. It traverses the linked list from front to rear using an iterator pointer, appending each car's ID (formatted with width 2) followed by a separator to a string stream, returning the complete representation.

Destructor

The destructor implements comprehensive cleanup of all dynamically allocated Car objects to prevent memory leaks. It iterates through the linked list using the front pointer, storing each node temporarily, advancing the pointer, and deleting the node, finally setting rear to nullptr for safety.

`Parkinglot` Class

The Parkinglot class serves as the central controller and main interface for the entire parking management system, orchestrating interactions between the waiting queue and multiple parking stacks. It implements core business logic for vehicle arrival, parking assignment, retrieval, and reorganization within a simulated parking facility with comprehensive operations.

`Q` (MyQueue Object)

This public MyQueue instance represents the waiting area where newly arriving vehicles line up before parking assignment. Operating on FIFO principles with bounded capacity, it ensures fair ordering based on arrival time and serves as the primary buffer between vehicle arrivals and parking space availability.

`parkings` (Vector of MyStack Objects)

This public vector container holds all individual parking stacks within the facility, each representing a distinct parking area operating on LIFO principles. The vector size is determined during construction, with each stack having uniform capacity limits for vehicle storage.

`display` Method

This method generates a comprehensive visual representation of the current system state for terminal-based monitoring. It outputs queue contents via Q.print() followed by each parking stack's contents via iterative print() calls, providing formatted runtime visibility during development and testing.

Constructor

This constructor initializes a complete parking lot system with specified capacity parameters for all components. It sets queue capacity directly, then creates the specified number of MyStack objects with uniform capacities, appending them to the parkings vector to establish the parking facility structure.

`initialize` Method

This method serves as the automatic parking assignment engine, transferring vehicles from the waiting queue into available parking spaces. It operates in a loop while the queue isn't empty, attempting to insert each front vehicle and dequeuing on success, returning 0 if the queue empties or 1 if parking becomes full first.

`find` Method

This method performs a comprehensive search across all parking stacks to locate a vehicle by its unique identifier. It returns a pair containing stack index and position within that stack (0 for top), or (-1, -1) if not found, using nested loops to traverse each stack's linked structure from top to bottom.

`isInQueue` Method

This method determines whether a vehicle with a given identifier is currently in the waiting queue. It checks for empty queue, then iterates through the linked structure up to capacity times, comparing IDs at each node and returning true upon match or false after exhaustion.

`addToQueue` Method

This method handles complete new vehicle arrival with validation, queue insertion, and automatic parking assignment. It checks for duplicate IDs in both parking and queue, attempts enqueue with provided parameters, calls initialize() for parking assignment, and returns success/full/duplicate status codes.

`insert` Method

This method places a vehicle into the first available parking space using sequential search strategy. It iterates through the parkings vector, attempting to push the car onto each stack in order, returning 0 on first success or 1 if all stacks are full after complete iteration.

`insertAt` Method

This method attempts to park a vehicle in a specific, user-designated parking stack by index. It directly calls push() on the specified stack with provided parameters, returning the boolean result (0 for success, 1 for stack full) without sequential searching, enabling directed parking assignments.

`sort` Method

This method delegates sorting of a specific parking stack to the corresponding MyStack object's internal sorting algorithm. It accepts a stack index parameter and calls sort() on that stack, triggering merge sort rearrangement of vehicles in ascending ID order within that parking area.

`move` Method

This method implements complex multi-vehicle relocation between parking stacks with intelligent overflow handling. It moves cars from source to destination stack, skipping full stacks and avoiding the source, wrapping around when reaching vector end, returning 0 if source empties or 1 if insufficient space.

`popCar` Method

This method handles specific vehicle departure from the parking lot, only if immediately accessible at the top of its stack. It locates the vehicle via find(), validates top position, pops it from the appropriate stack, calls initialize() to fill the vacated space, and returns success/blocked/not-found status codes.

`main.cpp` GUI Implementation

Overview and Structure

This file implements the graphical user interface for the Parking Management System using FLTK 1.3.x. The interface provides seven functional sections for system interaction: parking lot creation, vehicle entry, stack sorting, inter-stack movement, vehicle location, vehicle departure, and system visualization. The GUI follows a manual grid layout with consistent theming and real-time state updates.

Global Variables

The system maintains a global `Parkinglot` object initialized with default parameters (queue capacity 10, 3 stacks, stack capacity 5). Various FLTK widget pointers are declared globally for input fields and display areas, including queue configuration inputs, vehicle entry fields, operation-specific inputs, and output displays for system state and search results.

Helper Functions

`update_display()` refreshes the main visualization panel by constructing a string representation of queue and parking stack contents. It traverses the queue linked list and each stack's structure, formatting vehicle IDs and empty slots. `themed_button()` creates consistently styled buttons with blue backgrounds, white text, and rounded corners. `centered_btn_y()` calculates vertical button positioning.

Callback Functions

Seven callback functions handle user interactions: `create_parkinglot_cb()` reinitializes the system with user-specified parameters; `enter_parking_cb()` processes new vehicle arrivals; `sort_parking_cb()` triggers stack sorting; `move_parking_cb()` executes inter-stack vehicle movement; `find_parking_cb()` locates vehicles and displays results; `exit_parking_cb()` handles vehicle departures; `quit_cb()` terminates the application.

GUI Layout and Initialization

The main window is 1000x950 pixels with dark mode theming. Layout constants define row heights and component widths. The interface is constructed section-by-section using manual positioning: system creation inputs at the top, followed by vehicle entry, sorting controls, movement parameters, search functionality, departure controls, and quit button. A large display area on the right shows real-time system state.

Event Loop and Operation

After widget creation and callback binding, the window displays and enters FLTK's event loop. The `update_display()` function is called initially and after each user action to maintain synchronization between the visual interface and underlying system state. The application remains responsive to user interactions through FLTK's event-driven architecture.