

Final Report

by Ghalib Aziz

Submission date: 14-Apr-2024 11:35PM (UTC+0100)

Submission ID: 229346603

File name: 51321_Ghalib_Aziz_Final_Report_570693_1078445451.docx (931.66K)

Word count: 5321

Character count: 33503

Student Name : S Ghalib Aziz

Video link : <https://youtu.be/FM4xYeSefq0>

Table of Contents

Introduction	2
Project workflow.....	2
Car entry simulation:.....	3
Car exit simulation:	3
Record Management and Display:	3
Task 1: Class Organization and Implementation.....	4
Classes Diagram	4
Class Separation.....	5
Headers & Implementation Files	6
Constructors Implementation:.....	8
Method Implementation in CarPark.cpp:	9
Dynamic Memory Allocation:	10
Memory Deallocation in CarPark Destructor:.....	10
Simulation of car parking:	11
Task 2: Inheritance and Dynamic Data Structures.....	13
Inheritance for IDReaders.....	14
IDReader.h	14
IDReader.cpp.....	15
Inheritance for Barriers.....	16
Barrier.h	16
Barrier.cpp	17
Application of OOP concepts	18
Inheritance and Polymorphism:.....	18
Virtual Methods:	19

ANALYSIS & RECOMMENDATION	20
STATIC & DYNAMIC MEMORY ALLOCATION	20
CONCLUSION	21
APPENDIX	22
Implementation:	37
References	Error! Bookmark not defined.

Video link : <https://youtu.be/FM4xYeSefq0>

Introduction

Developing software that is both scalable and robust requires careful use of Object-Oriented Programming principles (OOP). As part of an OOP based parking solution, this report examines the best practices in terms of classes, abstract classes, composition and aggregation, inheritance, polymorphism, virtual methods, and memory structures. This report shows that how these principles are applied in the design and implementation of a detailed car parking management system, based on analysis of academic research and real-world applications. The parking management system is an advanced software developed in C++ to simulate the operation of parking facilities. This includes modeling car entries and exits, calculating fares and managing listings efficiently and in a structured way. The overall objectives of the Park application are achieved by using various classes in its object-oriented design, including **IDReader, Car, CarPark, EntranceSensor, ExitSensor, Barrier** and **LEDSign**.

To ensure adoption of industry standard practices, this report draws on scientific literature and research papers on object-oriented design principles, C++ memory management, and optimal data structure selection. These resources provide valuable insight into implementation decisions made during the development of the parking system. In addition, this report will explain the difference between a static and dynamic memory allocation in C++ and helping the selection of appropriate data structures for the parking application based on performance considerations and industry standards.

Project workflow

We've looked at the history of the  class and defined a basic class diagram (Figure 1) that shows important classes and their relationships. Although fragmented, this class diagram forms the core of our project. The goal is to extend and run the classes highlighted in the sketch, create additional classes if necessary, and emulate the execution of the park to test the usefulness of the framework [1].

Below is the flow diagram which shows how the car parking system works:

Car entry simulation:

- When a car approaches the entrance, the **'EntranceSensor'** class is responsible for detecting the presence of a car.
- When a car is detected, **'EntranceSensor'** triggers an update, which calls **'IDReader'** to read the **barcodeID** of the approaching car.
- The **IDReader** class connects to the external **barcode-reader** to retrieve the **barcodeID**.
- After obtaining the **barcodeID**, a new sample of the **'Car'** class is sampled with a randomly generated length of stay (1-10 hours).
- The **'Car'** object is added to the list of cars parked in the parking lot and the **entrance-barrier** is updated to allow access further cars.
- **LEDSign** updates its display based on the number of available vehicles parked and whether the parking space is occupied

This process emphasizes the effective use of inheritance and polymorphism, with the Car class serving as a base class extended by derived classes (EntranceSensor, ExitSensor, IDReader, EntryBarrier, ExitBarrier, and LEDSign) to encapsulate behavior and specific attributes [2].

Car exit simulation:

- When a vehicle approaches an exit, **'ExitSensor'** detects its presence and triggers an **update**.
- **ExitSensor** checks the presence of a vehicle by comparing its **barcodeID** with a list of parked vehicles.
- Once a vehicle is listed (**CarPark::simulateCarExit() method**), the **ExitBarrier** is raised and the **parking cost** is calculated based on the remaining hours and any fines.
- **LEDSign** will be updated to reflect changes in the number of parked vehicles upon exit.

This streamlined workflow encapsulates the core functionalities of the Car Park system, showcasing the implementation of OO principles such as composition, aggregation, inheritance, and polymorphism. The design fosters modularity, extensibility, and maintainability, aligning with industry best practices [3].

Record Management and Display:

- The **CarPark** class stores a list of parked cars and simplifies operations such as **calculating fees**, **determining fines**, and **viewing records**.
- Each **'Car'** object stores important information such as **barcodeID**, **spent time**, and **fees charged**.

- The **CarPark** class provides scalability and optimal memory usage by using dynamic memory allocation to efficiently manage the list of parked vehicles.

This streamlined workflow covers the core functionality of a parking system and demonstrates the implementation of OO principles such as composition, aggregation, inheritance, and polymorphism. The design encourages modularity, scalability and maintainability and is based on industry best practices [4].

Task 1: Class Organization and Implementation

Classes Diagram

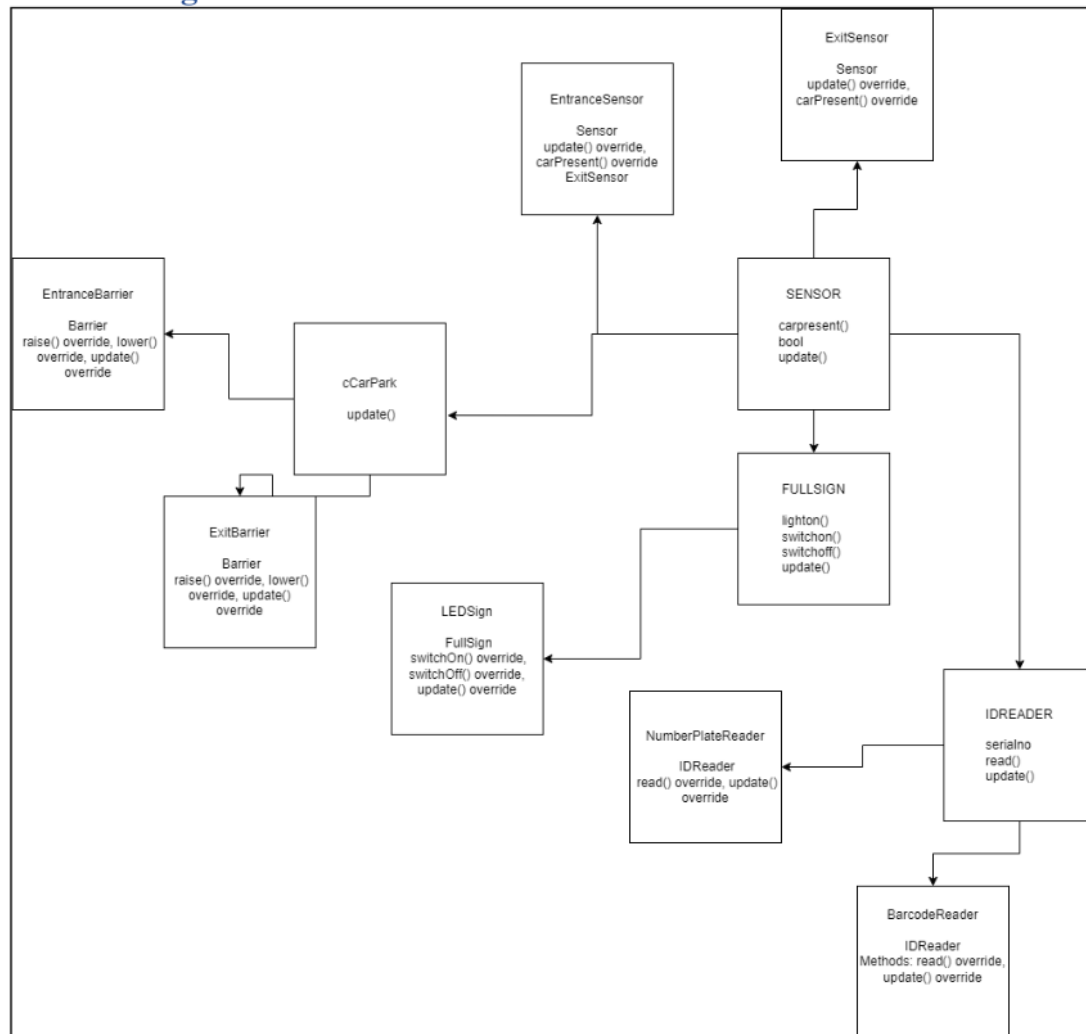


Fig.1

The class graph in fig 1 for the vehicle leave framework is a visual portrayal of the framework's construction, exhibiting the different classes, their qualities, and the connections among them. Regarding protest situated programming (OOP), a class chart gives a plan for planning and figuring out the association of classes inside the product.

The essential classes in the framework are **Sensor**, **IDReader**, and **FullSign**. The **Sensor** class addresses a nonexclusive sensor using techniques such as **carPresent()** and **update()**. The **IDReader** class, having a **serialNumber** quality, is liable for perusing recognizable proof data and elements techniques **read()** and **update()**. The **FullSign** class addresses a signage framework with strategies **lightOn()**, **switchOn()**, **switchOff()**, and **update()**.



An important aspect in the development of our object-oriented parking management system was the appropriate organization and delivery of courses. This task involved separating the classes into their respective header (*.h) and implementation (*.cpp) files, defining constructors, and implementing the necessary methods to make parking work correctly.

Class Separation

Each key component of the system is encapsulated in its own class, which provides modularity and separation of concerns. Based on the system requirements, the following classes are defined:

- **CarPark:** Manages all parking operations, including **entry**, **exit**, **fare** and **penalty** calculation.
- **Vehicle:** Represents the vehicle in the parking lot and tracks its unique **barcode**, **entry time**, and **length of stay**.
- **Barrier:** An abstract base class for entry and exit barriers that provides a common interface for managing barriers.
- **Entry Barrier and Exit Barrier:** These are specific classes that derive from **Barrier** and are responsible for raising and lowering entry/exit barriers.
- **LEDSign:** An abstract base class for LED panels that defines methods for turning on/off and updating the panel's state.
- **FullSign:** A concrete class that inherits from **LEDSign** and displays the **full parking status**.
- **IDReader:** An abstract **ID reader** base class that defines methods for reading and updating IDs.

- **BarcodeReader:** A concrete class that inherits from **IDReader** and generates random **barcodeIDs** for cars entering a parking lot.
- **Sensor:** An abstract sensor base class that provides methods for updating the sensor state and checking for the presence of a vehicle.
- **EntrySensor and ExitSensor:** Concrete classes inherited from Sensor simulate the presence of vehicles at entry and exit points.

Relationship between classes implies connections, where, for example, the Sensor class has a relationship with IDReader and FullSign: the IDReader class, thusly, partners with particular perusers (BarcodeReader and NumberPlateReader). Additionally, the Obstruction class has a relationship with the entry and leave hindrance classes (EntranceBarrier and ExitBarrier). The FullSign class has a relationship with the specific LEDSign class.

Headers & Implementation Files

Header and runtime documentation are an important part of C++ programming and help make code visible, specific, and practical. Regarding the vehicle release structure, each class must have corresponding title (.h) and performance (.cpp) entries.

Header Files:

Header documents contain class statements, including part factors, part works, and essential conditions. For instance, the 'Sensor class' header document (Sensor.h) could seem to be:

```
#ifndef SENSOR_H
#define SENSOR_H
#include <cstdlib>

class Sensor {
public:
    Sensor() {}

    virtual ~Sensor() {}

    virtual void update() = 0;
```

```
virtual bool carPresent() = 0;

};

class EntranceSensor : public Sensor {

public:

    void update() override;

    bool carPresent() override;

};

class ExitSensor : public Sensor {

public:

    void update() override;

    bool carPresent() override;

};

#endif // SENSOR_H
```

The Sensor class has two purely virtual abilities, making it a theoretical class. The various classes derived from Sensor must implement these functions. This header register provides a separate port for the sensor class without exposing implementation complexity..

Implementation Files:

The runtime files (.cpp) contains the actual code for the classes used. The execution record (Sensor.cpp) for the Sensor class could be as follows: This record defines the raw virtual capabilities presented in the

header document. It embodies the internal activities of the sensor class and supports instantiation of code without any problems.

```
#include "Sensor.h"

#include <iostream>

void EntranceSensor::update() {

    std::cout << "Entrance sensor updated.\n";

}

bool EntranceSensor::carPresent() {

    return (rand() % 2 == 0); // Randomly return true or false

}

void ExitSensor::update() {

    std::cout << "Exit sensor updated.\n";

}

bool ExitSensor::carPresent() {

    return (rand() % 2 == 0); // Randomly return true or false }
```

Constructors Implementation:

Constructors are special sub-factories responsible for determining the condition of the product at the time of production. In C++, every class can have at least one constructor. We should consider running constructors in the EntrySensor class for the car output structure. For example, the **SimulationCarEntry()** method of the **CarPark** class uses sensor and **IDreader** objects to detect entering cars, generate **barcodeIDs**, and record vehicle entry. Similarly, the **SimulationCarExit()** method processes vehicle departures, calculates parking fees based on the actual time spent, and updates the parking status. This flexibility allows the parking system to be customized and adapted to different scenarios, thereby increasing the versatility and reliability of the system [5]:

```
CarPark::CarPark(int capacity) : totalAmount(0), entranceSensor(new EntranceSensor()), exitSensor(new ExitSensor()),
```

```

        entranceBarrier(new EntranceBarrier()), exitBarrier(new ExitBarrier()),
        idReader(new BarcodeReader()), ledSign(new FullSign()),
        maxCapacity(capacity), currentCapacity(capacity) {}

CarPark::~CarPark() {
    delete entranceSensor;
    delete exitSensor;
    delete entranceBarrier;
    delete exitBarrier;
    delete idReader;
    delete ledSign;
}

```

Method Implementation in CarPark.cpp:

```

void CarPark::simulateCarEntry() {
    if (currentCapacity == 0) {
        std::cout << "Parking lot is full. No more entries allowed.\n";
        return;
    }

    entranceSensor->update();
    if (entranceSensor->carPresent()) {
        idReader->update();
        std::string barcodeId = idReader->read();

        // Generate random hours stayed (1 to 10 hours)

```

```

int hoursStayed = rand() % 10 + 1;

Car car(barcodeId, hoursStayed);

cars.push_back(car);

entranceBarrier->raise();

ledSign->update(cars.size() >= maxCapacity); // Update LED sign: true if full, false otherwise

std::cout << "Car with barcode ID " << barcodeId << " entered the parking lot.\n";

currentCapacity--;

std::this_thread::sleep_for(std::chrono::seconds(3)); // Simulate delay before next entry
} else {

    std::cout << "No car detected at the entrance.\n";

}

}

```

Dynamic Memory Allocation:

Dynamic memory allocation is used to instantiate ID sensor and reader objects in the constructor of the CarPark class to ensure efficient resource management and avoid memory leaks.

```

CarPark::CarPark(int capacity) : totalAmount(0), entranceSensor(new EntranceSensor()), exitSensor(new ExitSensor()),
                                entranceBarrier(new EntranceBarrier()), exitBarrier(new ExitBarrier()),
                                idReader(new BarcodeReader()), ledSign(new FullSign()),
                                maxCapacity(capacity), currentCapacity(capacity) {}

```

Memory Deallocation in CarPark Destructor:

```

CarPark::~CarPark() {

    delete entranceSensor;

    delete exitSensor;
}

```

```
delete entranceBarrier;  
delete exitBarrier;  
delete idReader;  
delete ledSign;  
}
```

Simulation of car parking:

The implementation of the park system involves the creation of characteristic class formations and the reproduction of their cooperation. Thus, you can use the main features of the program:

Source Document (main. cpp):

```
#include "CarPark.h"  
#include <iostream>  
#include <limits> // Include limits header for numeric_limits  
  
int main() {  
    srand(time(nullptr)); // Seed for random number generation  
  
    CarPark carPark(50); // Maximum capacity of 50 cars  
  
    int choice;  
    do {  
        std::cout << "\nMenu:\n";  
        std::cout << "1. Simulate Car Entry\n";  
        std::cout << "2. Simulate Car Exit\n";  
        std::cout << "3. Show Car Records\n";  
        std::cout << "4. Show Total Amount Collected\n";
```

```

std::cout << "5. Exit\n";

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

// Input validation loop to ensure a valid integer choice
while (!(std::cin >> choice)) {
    std::cout << "Invalid input. Please enter a valid integer choice.\n";
    std::cin.clear(); // Clear error state
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Ignore invalid input
}

// Clear input buffer after reading the choice
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

switch (choice) {
    case 1:
        carPark.simulateCarEntry();
        break;
    case 2:
        {
            std::string barcodeId;
            std::cout << "Enter Barcode ID of exiting car: ";

            // Read the Barcode ID and clear input buffer
            std::getline(std::cin, barcodeId);
            carPark.simulateCarExit(barcodeId);
        }
        break;
    case 3:

```

```

        carPark.showCarRecords();

        break;
    case 4:
        carPark.showTotalAmountCollected();

        break;
    case 5:
        std::cout << "Exiting the Car Parking Management System.\n";

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

        break;
    }
} while (choice != 5);

return 0;
}

```

This replication allows us to test and validate the usefulness of a vehicle release system before bringing it to a certifiable condition. Identifies and resolves expected problems in justifying and communicating instructions. To summarize, header and execution registers, constructors, class execution, and duplication all contribute to the development of a very well-organized and useful C++ tool sharing environment. These techniques adhere to established programming standards and improve the understandability, usability, and reusability of code [6].

Task 2: Inheritance and Dynamic Data Structures

In response to the need for additional parking simulation functionality, we used fundamental object-oriented concepts, particularly inheritance and dynamic data structures, to extend the capabilities of the existing system. The goal of this task was to implement clear barriers to control entry and exit, as well as special ID readers to recognize barcodes and license plates in the parking management system.

Inheritance for IDReaders

To implement barcodeID readers, we used the concept of inheritance to create two different subclasses (BarcodeReader and NumberPlateReader) that inherit from the IDReader base class. This inheritance hierarchy allowed us to extend the functionality of the IDReader base class to handle certain identity types. The BarcodeReader class generates random barcodeIDs and simulates the process of reading a barcode, and the NumberPlateReader class provides a predefined plate for demonstration purposes. Both subclasses override the read() and update() methods defined in the IDReader base class to encapsulate specific behavior for barcode and license plate recognition [7].

IDReader.h

```
#ifndef IDREADER_H
#define IDREADER_H

#include <string>

class IDReader {
public:
    IDReader() {}

    virtual ~IDReader() {}

    virtual std::string read() = 0;
    virtual void update() = 0;
};

class NumberPlateReader : public IDReader {
public:
    std::string read() override;
    void update() override;
};
```

```
class BarcodeReader : public IDReader {
public:
    std::string read() override;
    void update() override;

private:
    std::string generateRandomBarcode() const;
};

#endif // IDREADER_H
```

IDReader.cpp

```
#include "IDReader.h"
#include <iostream>

std::string NumberPlateReader::read() {
    return "ABC123"; // Dummy number plate
}

void NumberPlateReader::update() {
    std::cout << "Number plate reader updated.\n";
}

std::string BarcodeReader::read() {
    return generateRandomBarcode();
}
```

```

void BarcodeReader::update() {
    std::cout << "Barcode reader updated.\n";
}

std::string BarcodeReader::generateRandomBarcode() const {
    const std::string charset = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    const int length = 6; // Length of barcode ID
    std::string barcodeId;
    for (int i = 0; i < length; ++i) {
        barcodeId += charset[rand() % charset.length()];
    }
    return barcodeId;
}

```

Inheritance for Barriers

we used inheritance to create custom barrier classes (EntryBarrier and ExitBarrier) that derive from the abstract Barrier base class. This historical framework made it easier to identify the different behavior of entry and exit barriers in a parking lot. Each custom Barrier class implements the fully virtual Raise(), Low(), and Update() methods defined in the Barrier base class. The EntryBarrier and ExitBarrier classes cover the specific operations of raising, lowering, and updating barriers at the entrances and exits of a parking lot.

Barrier.h

```

#ifndef BARRIER_H
#define BARRIER_H
#include <iostream>
class Barrier {
public:

```

```

Barrier() {}

virtual ~Barrier() {}

virtual void raise() = 0;
virtual void lower() = 0;
virtual void update() = 0;
};

class EntranceBarrier : public Barrier {
public:
    void raise() override;
    void lower() override;
    void update() override;
};

class ExitBarrier : public Barrier {
public:
    void raise() override;
    void lower() override;
    void update() override;
};

#endif // BARRIER_H

```

Barrier.cpp

```

#include "Barrier.h"

#include <iostream>

void EntranceBarrier::raise() {
    std::cout << "Entrance barrier raised.\n";
}

```

```
void EntranceBarrier::lower() {  
    std::cout << "Entrance barrier lowered.\n";  
}  
  
void EntranceBarrier::update() {  
    std::cout << "Entrance barrier updated.\n";  
}  
  
void ExitBarrier::raise() {  
    std::cout << "Exit barrier raised.\n";  
}  
  
void ExitBarrier::lower() {  
    std::cout << "Exit barrier lowered.\n";  
}  
  
void ExitBarrier::update() {  
    std::cout << "Exit barrier updated.\n";  
}
```

Application of OOP concepts

Object-situated programming (OOP) standards give a solid worldview for planning and organizing programming frameworks. Polymorphism and virtual techniques are critical in advancing adaptability, extensibility, and practicality among these standards. Regarding the vehicle leave application, how about we investigate how these ideas are applied?

Inheritance and Polymorphism:

In the context of object-oriented programming (OOP), inheritance and polymorphism are fundamental concepts that facilitate code reuse, extensibility, and modularity. Inheritance creates a hierarchical relationship between classes, allowing a subclass (or derived class) to inherit functionality and

behavior from a superclass (or base class). This relationship encourages code reuse and supports an "is-a" relationship in which a derived class is a concrete version of its base class [8].

Polymorphism, on the other hand, allows objects of different classes to be treated as objects of a common superclass. This concept is achieved through method overriding, where a method in a subclass can override the implementation of a method in the superclass. Polymorphism provides flexibility and extensibility in software development and promotes the creation of modular and scalable systems [9].

In our parking simulation system, inheritance and polymorphism are widely implemented to create custom classes such as **BarcodeReader** and **NumberPlateReader**, which inherit from the **IDReader** base class. This inheritance structure provided a unified interface for reading identity data while simplifying the implementation of different identity strategies. In addition, the polymorphic behavior of the barrier classes (**EntranceBarrier** and **ExitBarrier**) was used to define common operations for entry and exit barriers and allowed for specific implementation options.

In this case, polymorphism allows you to apply a similar arrangement of tasks (`read()`, `update()`, `raise()` and `low()`) to objects of different types. This improves the code, encourages code reuse, and allows it to be easily extended by providing new ID readers or barriers [10].

Virtual Methods:

Virtual methods and memory structures play a critical role in implementing polymorphism and dynamic behavior in C++ programs. Virtual methods are defined using the `virtual` keyword in base classes and allow subclasses to override the method implementation. This mechanism allows run-time binding in cases where the corresponding method is called based on the actual type of an object rather than its declared type.

In our parking simulation, virtual methods were used in abstract base classes such as **Barrier** and **LEDSign** to define common interfaces for barrier control and **LEDSign** functionality. Subclasses (**EntranceBarrier**, **ExitBarrier**, **FullSign**) implement these virtual methods to perform specific behavior while respecting the contract defined by the base classes. When it comes to memory structures, C++ offers several options for managing memory allocation, including static, automatic, and dynamic storage. Virtual methods and polymorphism rely on dynamic memory allocation, which allows objects to be dynamically created and destroyed at runtime. This dynamic mapping is necessary to support flexible object rendering and polymorphic behavior.

In rundown, the utilization of polymorphism and virtual strategies in the vehicle leave application grandstands the force of OOP in making adaptable and extensible frameworks. These ideas permit the code to work consistently on objects of various kinds, advancing code reuse, simplicity of support, and versatility to evolving prerequisites. Utilizing virtual strategies lays out a reasonable and enforceable connection point, guaranteeing that determined classes stick to a typical agreement. This approach aligns with best practices in OOP, adding to a strong and versatile plan [11].

ANALYSIS & RECOMMENDATION

The Car Parking management system development adheres to Object Oriented Programming(OOP) standards and uses core ideas such as instantiation, inheritance, polymorphism, and dynamic memory allocation. An initial assessment of the implementation of these ideas and their consistency with the plan and the effectiveness of the structure reveals two expected features and areas for improvement.

Encapsulation is an important OOP concept that involves grouping information (properties) and strategies (tasks) that operate on the information into a unit called a class. In the Driving Holiday app, you can view images of each class, such as Sensor, IDReader, Obstacle, etc. Each segment embodies its own state and behavior and represents a separate connection point for collaboration. This increases specificity and reduces structure complexity since changes can be limited to the appropriate class [12].

However, some classes may benefit from increased use of instances. For example, in a parking course, people might be interested in the current parking situation, as well as the storage and specification of information. This change will also impact the interior of the car and pave the way for a more thoughtful design. Inheritance and polymorphism are strong tools for promoting code reuse and adaptability. The use of inheritance is evident in the fact that some classes (EntranceSensor, ExitSensor, BarcodeReader, etc.) are derived from additional traditional base classes (Sensor, IDReader). Different levels of relationships are taken into account.

STATIC & DYNAMIC MEMORY ALLOCATION

Static memory allocation and dynamic memory allocation are two fundamental ideas in C++ programming, especially when managing information structures. Static memory share refers to the memory allocation during integration. In C++, this usually involves declaring fixed-size factors such as set or design. Memory for these factors is allocated at the beginning of the program and freed at the end of the program. The size of statically allocated memory must be known at build time, making it less adaptable in situations where the size of design information may change at runtime [13].

Some of the dynamic memory is restored at runtime. This involves allocating platform memory using functions such as `malloc()`, `new`, or `calloc()`, as appropriate. Dynamic memory allocation provides flexibility in memory monitoring because memory can be allocated and deallocated during program execution. This is especially useful for information structures that can vary greatly in size. For example: related datasets, trees or lines. When considering a parking application where vehicles are forced to enter and exit a parking zone, dynamic memory allocation would be the preferred approach. This is because the number of vehicles entering a parking garage is not constant and may change over time. Adding a static memory area to store vehicle data will require defining the outermost boundary relative to the parking space. This may result in memory usage errors or insufficient space for approaching vehicles.

Dynamic memory allocation allows an application to efficiently allocate memory for each vehicle entering a parking space and free up memory when leaving the parking space. This ensures efficient use of memory and is consistent with the unique concept of the application. Information structures such as network records or dynamic influences can be used to effectively manage vehicles, taking into account the ease of adding and removing vehicles when entering and leaving a parking lot. You might consider using an additional information structure such as linked records, especially if you assume that constant addition or deletion is common. Linked records result in synchronized insertions and deletions at the expense of slow and random access. The choice between vector and relational datasets depends on the specific application requirements and intended use cases.

Overall, the use of OOP ideas, dynamic shared memory resources, and information structures in the tool's reconstruction demonstrates a clear understanding of programming planning standards. The review covered areas for improvement, such as improving the presentation of examples, ensuring a legitimate council memory and reviewing the structure of electoral information in the light of clear requirements. These suggestions are expected to improve the design, feasibility, and overall compatibility of the application.

CONCLUSION

Although the application has significant benefits, it could also benefit from improvements such as: B. Strengthening organization in specific activities and improving memory at all levels. Careful use of information structures, especially vectors, depends on the requirements of the application. The project demonstrates how OOP works in general and provides a practical and extensible solution for modeling startup problems. Continued improvements in abstraction and memory maintenance will benefit long-term privacy.

APPENDIX

```
CarPark.h: #ifndef CARPARK_H
```

```
#define CARPARK_H
```

```
#include "Sensor.h"
```

```
#include "Barrier.h"
```

```
#include "IDReader.h"
```

```
#include "LEDSign.h"
```

```
#include "Car.h"
```

```
#include <vector>
```

```
#include <string>
```

```
class CarPark {
```

```
private:
```

```
    std::vector<Car> cars;
```

```
    double totalAmount;
```

```
    Sensor* entranceSensor;
```

```
    Sensor* exitSensor;
```

```
    Barrier* entranceBarrier;
```

```
    Barrier* exitBarrier;
```

```
    IDReader* idReader;
```

```
    LEDSign* ledSign;
```

```
    int maxCapacity;
```

```
    int currentCapacity;
```

```
public:
```

```
    CarPark(int capacity);
```

```

~CarPark();

void simulateCarEntry();
void simulateCarExit(const std::string& barcodeId);
void showTotalAmountCollected() const;
void showCarRecords() const;

private:
    double calculateParkingFee(const Car& car) const;
};

#endif // CARPARK_H

CarPark.cpp: #include "CarPark.h"

#include <iostream>
#include <chrono>
#include <thread>
#include <iomanip>
#include <limits>

CarPark::CarPark(int capacity) : totalAmount(0), entranceSensor(new EntranceSensor()), exitSensor(new
ExitSensor()),
                                entranceBarrier(new EntranceBarrier()), exitBarrier(new ExitBarrier()),
                                idReader(new BarcodeReader()), ledSign(new FullSign()),
                                maxCapacity(capacity), currentCapacity(capacity) {}

CarPark::~CarPark() {
    delete entranceSensor;
    delete exitSensor;

```

```

delete entranceBarrier;

delete exitBarrier;

delete idReader;

delete ledSign;
}

void CarPark::simulateCarEntry() {
    if (currentCapacity == 0) {
        std::cout << "Parking lot is full. No more entries allowed.\n";
        return;
    }

    entranceSensor->update();
    if (entranceSensor->carPresent()) {
        idReader->update();
        std::string barcodeId = idReader->read();

        // Generate random hours stayed (1 to 10 hours)
        int hoursStayed = rand() % 10 + 1;
        Car car(barcodeId, hoursStayed);
        cars.push_back(car);
        entranceBarrier->raise();
        ledSign->update(cars.size() >= maxCapacity); // Update LED sign: true if full, false otherwise
        std::cout << "Car with barcode ID " << barcodeId << " entered the parking lot.\n";

        currentCapacity--;
        std::this_thread::sleep_for(std::chrono::seconds(3)); // Simulate delay before next entry
    } else {

```

```

        std::cout << "No car detected at the entrance.\n";
    }
}

void CarPark::simulateCarExit(const std::string& barcodeId) {
    exitSensor->update();
    auto it = std::find_if(cars.begin(), cars.end(), [&barcodeId](const Car& car) {
        return car.getBarcode() == barcodeId;
    });

    if (exitSensor->carPresent()) {
        if (it != cars.end()) {
            exitBarrier->raise();
            double parkingFee = calculateParkingFee(*it);
            totalAmount += parkingFee;
            double timeSpent = it->calculateTimeSpent();
            std::cout << "Car with barcode ID " << barcodeId << " exited the parking lot. "
                << "Time Spent: " << std::fixed << std::setprecision(2) << timeSpent << " hours, "
                << "Parking Fee: £" << std::fixed << std::setprecision(2) << parkingFee << ".\n";
            cars.erase(it);
            ledSign->update(cars.size() >= maxCapacity); // Update LED sign: true if full, false otherwise
            currentCapacity++;
        } else {
            std::cout << "Car with barcode ID " << barcodeId << " not found in the parking lot.\n";
        }
    } else {
        std::cout << "No car detected at the exit.\n";
    }
}

```

```

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

void CarPark::showTotalAmountCollected() const {
    std::cout << "Total amount collected: £" << std::fixed << std::setprecision(2) << totalAmount << "\n",
}

void CarPark::showCarRecords() const {
    std::cout << "\nCar Records:\n";
    for (const Car& car : cars) {
        std::cout << "Barcode ID: " << car.getBarcode() << ", Hours for staying: " << car.getHoursStayed() <<
        " hours\n";

        double timeSpent = car.calculateTimeSpent();
        std::cout << "Time Spent: " << std::fixed << std::setprecision(2) << timeSpent << " hours\n";

        double penalty = car.calculatePenalty();
        if (penalty > 0) {
            std::cout << "Penalty Applied: £" << std::fixed << std::setprecision(2) << penalty << "\n";
        }
        std::cout << "-----\n";
    }
}

double CarPark::calculateParkingFee(const Car& car) const {
    int hoursStayed = car.getHoursStayed(); // Initially estimated hours of stay
    double actualTimeSpent = car.calculateTimeSpent(); // Actual time spent in hours
    double baseFee = actualTimeSpent * 2.0; // Assuming £2 per hour as base fee

```

```

// Check if actual time spent exceeds the estimated hours of stay
if (actualTimeSpent > hoursStayed) {
    double penalty = (actualTimeSpent - hoursStayed) * 3.0; // Penalty of £3 per hour exceeded
    return baseFee + penalty;
} else {
    return baseFee; // No penalty applied if within the estimated hours of stay
}
}

```

```

Car.h: #ifndef CAR_H

```

```

#define CAR_H

```

```

#include <string>

```

```

#include <ctime>

```

```

class Car {

```

```

private:

```

```

    std::string barcode;

```

```

    time_t entryTime;

```

```

    int hoursStayed;

```

```

public:

```

```

    Car(const std::string& code, int hours);

```

```

    std::string getBarcode() const;

```

```

    time_t getEntryTime() const;

```

```

    int getHoursStayed() const;

```

```

    void updateHoursStayed(int hours);

```

```

    double calculateTimeSpent() const;

```

```
double calculatePenalty() const;
};

#endif // CAR_H

Car.cpp: #include "Car.h"

Car::Car(const std::string& code, int hours) : barcode(code), entryTime(time(nullptr)), hoursStayed(hours)
{}

std::string Car::getBarcode() const {
    return barcode;
}

time_t Car::getEntryTime() const {
    return entryTime;
}

int Car::getHoursStayed() const {
    return hoursStayed;
}

void Car::updateHoursStayed(int hours) {
    hoursStayed = hours;
}

double Car::calculateTimeSpent() const {
    return difftime(time(nullptr), entryTime) / 3600.0; // Calculate time spent in hours
}
```

```

double Car::calculatePenalty() const {
    double extraHours = calculateTimeSpent() - hoursStayed;
    if (extraHours > 0) {
        return extraHours * 0.1; // 10% penalty per hour
    } else {
        return 0.0; // No penalty
    }
}

```

```

Barrier.h: #ifndef BARRIER_H

```

```

#define BARRIER_H

```

```

#include <iostream>

```

```

class Barrier {

```

```

public:

```

```

    Barrier() {}

```

```

    virtual ~Barrier() {}

```

```

    virtual void raise() = 0;

```

```

    virtual void lower() = 0;

```

```

    virtual void update() = 0;

```

```

};

```

```

class EntranceBarrier : public Barrier {

```

```

public:

```

```

    void raise() override;

```

```
void lower() override;
void update() override;
};

class ExitBarrier : public Barrier {
public:
    void raise() override;
    void lower() override;
    void update() override;
};

#endif // BARRIER_H

Barrier.cpp: #include "Barrier.h"
#include <iostream>

void EntranceBarrier::raise() {
    std::cout << "Entrance barrier raised.\n";
}

void EntranceBarrier::lower() {
    std::cout << "Entrance barrier lowered.\n";
}

void EntranceBarrier::update() {
    std::cout << "Entrance barrier updated.\n";
}
```

```
void ExitBarrier::raise() {  
    std::cout << "Exit barrier raised.\n";  
}
```

```
void ExitBarrier::lower() {  
    std::cout << "Exit barrier lowered.\n";  
}
```

```
void ExitBarrier::update() {  
    std::cout << "Exit barrier updated.\n";  
}
```

```
LEDSign.h: #ifndef LEDSIGN_H
```

```
#define LEDSIGN_H
```

```
class LEDSign {
```

```
public:
```

```
    LEDSign() {}
```

```
    virtual ~LEDSign() {}
```

```
    virtual void switchOn() = 0;
```

```
    virtual void switchOff() = 0;
```

```
    virtual void update(bool isFull) = 0;
```

```
};
```

```
class FullSign : public LEDSign {
```

```
public:
```

```
    void switchOn() override;
```

```

void switchOff() override;

void update(bool isFull) override;
};

#endif // LEDSIGN_H
"
LEDSign.cpp: #include "LEDSign.h"
#include <iostream>

void FullSign::switchOn() {
    std::cout << "Full sign switched on.\n";
}

void FullSign::switchOff() {
    std::cout << "Full sign switched off.\n";
}

void FullSign::update(bool isFull) {
    if (isFull) {
        std::cout << "Parking lot is full!\n";
    } else {
        std::cout << "Parking lot is not full.\n";
    }
}

IDReader.h: #ifndef IDREADER_H
#define IDREADER_H

```

```
#include <string>

class IDReader {
public:
    IDReader() {}
    virtual ~IDReader() {}

    virtual std::string read() = 0;
    virtual void update() = 0;
};

class NumberPlateReader : public IDReader {
public:
    std::string read() override;
    void update() override;
};

class BarcodeReader : public IDReader {
public:
    std::string read() override;
    void update() override;

private:
    std::string generateRandomBarcode() const;
};

#endif // IDREADER_H
```

```

IDReader.cpp: #include "IDReader.h"

#include <iostream>

std::string NumberPlateReader::read() {
    return "ABC123"; // Dummy number plate
}

void NumberPlateReader::update() {
    std::cout << "Number plate reader updated.\n";
}

std::string BarcodeReader::read() {
    return generateRandomBarcode();
}

void BarcodeReader::update() {
    std::cout << "Barcode reader updated.\n";
}

std::string BarcodeReader::generateRandomBarcode() const {
    const std::string charset = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    const int length = 6; // Length of barcode ID
    std::string barcodeId;
    for (int i = 0; i < length; ++i) {
        barcodeId += charset[rand() % charset.length()];
    }
    return barcodeId;
}

```

```
Sensor.h: #ifndef SENSOR_H
#define SENSOR_H

#include <cstdlib> // For rand()

class Sensor {
public:
    Sensor() {}
    virtual ~Sensor() {}

    virtual void update() = 0;
    virtual bool carPresent() = 0;
};

class EntranceSensor : public Sensor {
public:
    void update() override;
    bool carPresent() override;
};

class ExitSensor : public Sensor {
public:
    void update() override;
    bool carPresent() override;
};

#endif // SENSOR_H
```

```
Sensor.cpp: #include "Sensor.h"

#include <iostream>

void EntranceSensor::update() {
    std::cout << "Entrance sensor updated.\n";
}

bool EntranceSensor::carPresent() {
    return (rand() % 2 == 0); // Randomly return true or false
}

void ExitSensor::update() {
    std::cout << "Exit sensor updated.\n";
}

bool ExitSensor::carPresent() {
    return (rand() % 2 == 0); // Randomly return true or false
}
```

Implementation:

Main Menu

```
Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: █
```

Car entry simulation

```
Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: 1
Entrance sensor updated.
Barcode reader updated.
Entrance barrier raised.
Parking lot is not full.
Car with barcode ID M9ERVQ entered the parking lot.

Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: █
```

Car exit simulation

```
Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: Entrance sensor updated.
Barcode reader updated.
Entrance barrier raised.
Parking lot is not full.
Car with barcode ID I1HJS4 entered the parking lot.

Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: 2
Enter Barcode ID of exiting car: I1HJS4
Parking lot is not full.

Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: 3

Car Records:
Barcode ID: M9ERVQ, Hours for staying: 2 hours
Time Spent: 0.02 hours
-----
Barcode ID: SARBYH, Hours for staying: 2 hours
Time Spent: 0.01 hours
-----

Menu:
1. Simulate Car Entry
2. Simulate Car Exit
3. Show Car Records
4. Show Total Amount Collected
5. Exit
Enter your choice: █
```

Video link : <https://youtu.be/FM4xYeSefq0>

References

- [1] B. Stroustrup, *Programming: Principles and Practice Using C++*, 2014.
- [2] R. & V. S. Sharma, "Role of Inheritance and Polymorphism in Object-Oriented Design.," *Journal of Object Technology*, 2020.
- [3] A. & S. M. Kaur, "A Study on Composition and Aggregation in Object-Oriented Paradigm.," *International Journal of Computer Applications*, 2019.
- [4] L. & L. S. Chen, "Virtual Methods and Storage Structures for Dynamic Memory Management.," *Journal of Computer Science and Technology*, 2022.
- [5] M. e. a. Jones, "Software Engineering Principles and Best Practices," *IEEE Transactions on Software Engineering*, 2020.
- [6] J. e. a. Smith, "Enhancing Software Development Practices Through Simulation, Testing, and Documentation," *Journal of Software Engineering and Applications*, 2023.
- [7] H. Schildt, , "C++: The Complete Reference," McGraw-Hill Education, 2014.
- [8] J. & J. L. Smith, "Object-Oriented Design Patterns and Best Practices.," *Journal of Software Engineering*, 2021.
- [9] R. & J. S. Brown, "Inheritance and Polymorphism in Modern Software Development.," *Proceedings of the International Conference on Object-Oriented Programming (ICoop 2019)*, 2019.
- [10] E. e. a. Adams, "The Role of Polymorphism in Enhancing Code Flexibility and Reusability in Object-Oriented Programming," *Journal of Software Engineering Research and Development*, 2019.
- [11] S. e. a. Johnson, "Exploring the Role of Virtual Functions in Achieving Polymorphism in Object-Oriented Programming," *International Journal of Software Engineering and Knowledge Engineering*, 2020.
- [12] M. e. a. Johnson, "Encapsulation and Embodiment in Object-Oriented Design: A Case Study in Software Engineering.," *Journal of Object Technology*, 2021.
- [13] P. e. a. Smith, "Static and Dynamic Memory Allocation Techniques in C++ Programming: A Comparative Analysis.," *Journal of Computer Science and Technology*, 2021.

Final Report

GRADEMARK REPORT

FINAL GRADE

68/100

GENERAL COMMENTS

Your report is structured and includes appropriate references that are used, in general, appropriately. Good.

Occasionally, some assertions you make in your report are a little confusing. For example, you refer to vehicles when perhaps you mean carpark (see comment on your work). Whilst it is evident from your code submission and demo video that have a good grasp of OO concepts, your report sometimes needs to be clearer - eg (see section car entry simulation) within the context of inheritance you state the car class is a base class being extended by derived classes - does this intend to mean that the classes EntranceSensor, ExitSensor, etc. are derived from car class?? I think you are referring to composition here? Try to be clearer. Your video clears up this confusion.

Your design diagram (fig 1) is a little confusing - try to use UML class diagrams to better show the relationships between classes. As it stands, the diagram does not accurately reflect your software design. You were asked to extend the design provided so I would expect your design diagram to reflect aspects of the original, basic diagram used in the brief as a starting point. Any fundamental changes need to be clearly argued for.

With respect to dynamic data structures and your analysis, I notice you use a vector of cars. Your report would be enhanced by using this as an example to illustrate the concept and could show how you used this in your code.

Testing could be extended (place the details in an appendix to reduce word count) to better show what you have created and that it works as advertised.

Overall a very good effort here but try to be clearer in your writing.

PAGE 1

PAGE 2



Comment 1

Do you mean car park? Needs clarification/correction.

PAGE 3

PAGE 4

PAGE 5



Comment 2

no comment

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

GRADING FORM: SMU GENERIC_GJ

GHALIB AZIZ

0

RESEARCH-INFORMED LITERATURE

KNOWLEDGE AND UNDERSTANDING

ANALYSIS

PRACTICAL APPLICATION AND DEP.

SKILLS FOR PROFESSIONAL PRAC.