Object oriented Programming Final Exam

**Assignment 1**

1. What the program does:

Certainly! Here's a brief explanation of what each of the files, `Car.h`, `Car.cpp`, and `main\_race.cpp`, does:

1. Car.h:

`Car.h` is a header file that defines the `Car` class and its member functions. Here's what it does:

- It includes necessary header files, such as `<string>`, `<stdexcept>`, and `<iostream>`.

- It declares a custom exception class `CarException`, derived from `std::runtime\_error`, to handle exceptional conditions specific to the `Car` class.

- The `Car` class is defined, which represents a car with properties like speed, gear, and functionality to accelerate, brake, and shift gears. Key member functions include:

- `Car::Car()`: The default constructor initializes the car's speed and gear to zero.

- `Car::accelerate()`: Increases the car's speed by 10 units.

- `Car::brake()`: Decreases the car's speed by 10 units but doesn't allow it to go below zero.

- `Car::shiftGear(int gear)`: Changes the car's gear to the specified value.

- `Car::getSpeed()`: Returns the car's current speed.

- `Car::getGear()`: Returns the car's current gear.

- The `CarException` class provides a constructor that takes an error message and passes it to the base class constructor.

2. Car.cpp:

`Car.cpp` is the implementation file for the `Car` class. Here's what it does:

- It includes the `"Car.h"` header file to access the class definition and exception handling.

- It defines the member functions of the `Car` class, including the constructor and the functions for accelerating, braking, shifting gears, and retrieving speed and gear values.

- The member function implementations include logic to ensure that the car's properties, such as speed and gear, are updated correctly based on the specified operations.

3. main.cpp:

`main.cpp` is the main program file that demonstrates the functionality of the `Car` class through several test cases. Here's what it does:

- It includes necessary header files, such as `<iostream>`, `"Car.h"`, and `"Car.cpp"`, to access the `Car` class and its member functions.

- In the `main()` function, it defines three test cases to showcase different aspects of the `Car` class's functionality.

- Test Case 1 - Basic Functionality:

- It creates a dynamic `Car` object (`Car\* car1 = new Car();`) and demonstrates basic car functionality.

- It prints the car's initial speed, accelerates, applies the brake, and then deletes the car object.

- This test case showcases how to create and use a `Car` object, access its properties (speed and gear), and perform basic operations (accelerate and brake).

- Test Case 2 - Gear Shifting:

- Similar to Test Case 1, it creates another dynamic `Car` object (`Car\* car2 = new Car();`) and demonstrates gear shifting functionality.

- It accelerates, shifts gears, accelerates again, and then deletes the car object.

- This test case highlights the ability to change the car's gear and observe the effect on its speed.

- Test Case 3 - Exception Handling:

- Again, it creates a dynamic `Car` object (`Car\* car3 = new Car();`) and focuses on exception handling.

- It accelerates, applies the brake, and then attempts to apply the brake again, triggering an exception.

- The exception is caught using a `try-catch` block, and the error message is printed.

- This test case illustrates how exceptions are handled when attempting to perform invalid operations on a `Car` object.

- Each test case prints relevant information about the car's state and the operations performed.

- The program concludes by returning 0 if there are no exceptions; otherwise, it returns 1 in case of exceptions.

In summary, `main.cpp` serves as a driver program to demonstrate the usage of the `Car` class by creating objects, performing operations, and handling exceptions when necessary. It provides a clear and structured way to showcase the car's functionality and exception handling capabilities defined in the `Car` class.

1. The coding standards/best practices concepts used in the code:

1. Dynamic Memory Allocation (new and delete): The program uses dynamic memory allocation with `new` to create objects of the `Car` class. This allows objects to be created during runtime and explicitly destroyed using `delete` when they are no longer needed. This is in contrast to static memory allocation, where memory is allocated at compile-time.

2. Pointers: Pointers are used to manage the dynamically allocated objects. Pointers are variables that store memory addresses. In this program, `Car\*` pointers are used to point to dynamically allocated `Car` objects. Pointers enable access to the objects' member functions and data.

3. Inheritance: The program demonstrates the concept of inheritance by defining a derived class (`Car`) that inherits from a base class (`Vehicle`). Inheritance allows the derived class to inherit the data members and member functions of the base class. This promotes code reuse and supports the "is-a" relationship between classes.

4. Custom Exception Handling: A custom exception class, `CarException`, is defined by inheriting from the standard C++ `std::runtime\_error` class. Custom exception classes allow for more specific and meaningful error messages to be thrown and caught during runtime. This enhances the program's error handling capabilities.

5. Use of Standard Library Headers: The program includes standard library headers such as `<iostream>` for input and output, `<stdexcept>` for exception handling, and `<string>` (in the corrected code) for string manipulation. Using standard library headers provides access to pre-defined functionality and classes.

6. Error Messages: The program provides descriptive error messages when exceptions are thrown and caught. This makes it easier to understand and diagnose issues when they occur during program execution.

7. Makefile: A Makefile is used to automate the compilation process. Makefiles define rules for compiling source code files, specifying dependencies, and linking object files to create the final executable. This simplifies the build process and ensures that only necessary files are recompiled when changes are made.

8. Compilation Flags: The program uses compilation flags (`-Wall` and `-g`) to enable additional compiler warnings and debugging information. These flags help identify potential issues in the code and aid in debugging.

These concepts collectively contribute to creating a more robust and maintainable program that demonstrates object-oriented programming principles, dynamic memory management, and effective error handling.

**Assignment 2**

1. What the program does:

1. main\_race.cpp:

`main\_race.cpp` is the main program file that simulates a race scenario involving multiple `Car` objects. Here's what it does:

- It includes necessary header files, such as `"Car.h"`, `<vector>`, and `<algorithm>`.

- It defines a function `determineWinner` that takes a vector of `Car` objects and determines the winner based on their final speeds.

- In `main()`, it declares a constant `numCars` to specify the number of cars in the race and creates a dynamic array of `Car` objects using `std::vector<Car> cars(numCars)`.

- It defines five test cases that simulate different race scenarios, including acceleration, gear shifting, and braking operations for the cars.

- After simulating the race scenarios, it prints out the results, including the speeds and gears of each car at each time step and the winner of the race.

In summary, `Car.h` and `Car.cpp` define the `Car` class and its member functions, while `main\_race.cpp` uses these classes to simulate a race scenario with multiple cars, determining the winner based on their final speeds. The code leverages dynamic memory allocation through the use of `std::vector` to manage the cars and their properties.

1. The coding standards/best practices concepts used in the code:

Dynamic memory allocation is an essential concept in C++ that allows the user to allocate and deallocate memory for objects at runtime. In the provided code, dynamic memory allocation is used primarily for managing a dynamic array of `Car` objects and avoiding memory leaks. Here's how dynamic memory allocation is utilized in the code:

1. Dynamic Array of `Car` Objects:

In the code, a dynamic array of `Car` objects is created using `std::vector<Car> cars(numCars);`. Here's how dynamic memory allocation works in this context:

- `std::vector` is a container class that manages a dynamic array internally.

- The `numCars` variable determines the size of the dynamic array.

- Memory for the `Car` objects is allocated on the heap at runtime when the `std::vector` is constructed.

- Each `Car` object within the vector is created and initialized using its default constructor.

- Dynamic memory allocation allows the array to grow or shrink as needed without explicitly managing memory.

```cpp

const int numCars = 5; // Number of cars in the race

std::vector<Car> cars(numCars); // Create a vector of Car objects

```

2. Accessing Dynamic Objects:

You can access individual `Car` objects within the vector using array-like syntax. The dynamic memory management of the vector ensures that memory is allocated efficiently:

```cpp

cars[0].accelerate(); // Access and operate on the first Car object

```

3. No Manual Memory Management:

One significant advantage of using `std::vector` is that it automatically handles memory allocation and deallocation for you. When the vector goes out of scope or is explicitly destroyed, it deallocates the memory used by the contained `Car` objects.

```cpp

// No manual memory deallocation is needed

// Memory is automatically released when 'cars' goes out of scope

```

4. Avoiding Memory Leaks:

Dynamic memory allocation and deallocation help prevent memory leaks, which can occur if you allocate memory on the heap (using `new`) but forget to release it (using `delete`). By using containers like `std::vector`, you delegate memory management responsibilities to C++'s standard library, reducing the risk of memory leaks.

In summary, dynamic memory allocation in C++ is used to allocate memory on the heap at runtime, allowing you to work with objects whose size is determined dynamically. Containers like `std::vector` simplify the process of dynamic memory allocation and deallocation, reducing the likelihood of memory leaks and making your code more robust and maintainable.

programming concepts and techniques:

1. Vectors: The code uses the `std::vector` container from the C++ Standard Library to manage a dynamic array of `Car` objects. Vectors provide dynamic sizing and efficient memory management.

```cpp

std::vector<Car> cars(numCars);

```

2. Lambda Expressions: Lambda expressions are used in the `determineWinner` function to define custom comparison logic for finding the winning car based on speed.

```cpp

Car& determineWinner(std::vector<Car>& cars) {

Car\* winner = &cars[0];

for (size\_t i = 1; i < cars.size(); ++i) {

if (cars[i].getSpeed() > winner->getSpeed()) {

winner = &cars[i];

}

}

return \*winner;

}

```

The lambda expression `(const Car& car1, const Car& car2) { return car1.getSpeed() < car2.getSpeed(); }` is used as a custom comparator within `std::max\_element` to determine the winning car.

3. Iteration: The code includes `for` loops to iterate over cars, time steps, and perform various actions such as accelerating, braking, and shifting gears.

```cpp

for (int i = 0; i < numCars; ++i) {

for (int j = 0; j < 3; ++j) {

cars[i].accelerate();

}

}

```

4. Conditional Statements: Conditional statements (`if`) are used to control the behavior of cars during different time steps. For example, during the fifth time step, certain cars are instructed to brake, while others are told to accelerate.

```cpp

if (time == 5) {

cars[0].brake();

cars[2].brake();

cars[4].brake();

cars[1].accelerate();

cars[3].accelerate();

} else if (time > 5) {

if ((time - 5) % 2 == 0) {

cars[0].accelerate();

cars[2].accelerate();

cars[4].accelerate();

} else {

cars[0].brake();

cars[2].brake();

cars[4].brake();

}

}

```

5. Output Statements: The code uses `std::cout` to print messages and the state of each car during each time step.

```cpp

std::cout << "\nTime Step " << time + 1 << ":" << std::endl;

```

6. Comments: Comments are used throughout the code to provide explanations and document the purpose of various sections of the program.

These additional concepts and techniques help enhance the functionality and readability of the code, making it more comprehensive and maintainable.