Question 1:

We’re gonna be picking up where we left off in the last assignment: FAST’s Transportation System. For this scenario, you are required to revisit the system, and update it according to the newer concepts that we have discussed. It should include the following concepts: Inheritance, Constructor Chaining, Polymorphism (static/dynamic), Operator Overloading, etc. Things to consider: Introduce new classes, such as teacher and staff members who can also avail the transport. Is there a parent class that can encompass some of the functionalities for students, teachers and staff members? Are some functionalities different between users? For example, students pay on a semester-by-semester basis, and teachers pay on a monthly basis. Can you perform operator overloading to verify if two objects are the same or not? If so, demonstrate it in your classes. For example; are two routes the same? Feel free to refer to the question # 4 from the previous assignment for functionalities that you have used.

#include<iostream>

#include<string>

using namespace std;

// 24K-0010 WAMIZA NOMAN

class TransportationCard {

private:

int cardID;

bool isActive;

public:

TransportationCard(int id) : cardID(id), isActive(false) {}

void activateCard() {

isActive = true;

}

void deactivateCard() {

isActive = false;

}

bool checkCardStatus() const {

return isActive;

}

};

class User {

protected:

string name;

int userID;

bool hasPaidFees;

string assignedStop;

public:

User(string n, int id, string stop) : name(n), userID(id), assignedStop(stop), hasPaidFees(true) {}

virtual void payFees() = 0;

virtual void payFees(double discount) {

cout<<"Discounted fee payment not applicable."<<endl;

}

virtual void tapCard() const = 0;

virtual void displayInfo() const {

cout<<"ID: "<<userID<<endl<<"Name: "<<name<<endl<<"Stop: "<<assignedStop<<endl<<"Paid Fees: "<<((hasPaidFees)?"Yes":"No")<<endl;

}

virtual ~User() {}

};

class Student : public User {

private:

TransportationCard card;

public:

Student(string n, int id, string stop) : User(n, id, stop), card(id) {}

void payFees() override {

hasPaidFees = true;

card.activateCard();

cout<<name<<" has paid semester fees. Card activated."<<endl;

}

void payFees(double discount) override {

hasPaidFees = true;

card.activateCard();

cout<<name<<" has paid semester fees with a discount of "<<discount<<"%"<<endl;

}

void tapCard() const override {

if (card.checkCardStatus())

cout<<name<<" attendance recorded."<<endl;

else

cout<<"Card inactive. Pay semester fees to activate."<<endl;

}

};

class Teacher : public User {

public:

Teacher(string n, int id, string stop) : User(n, id, stop) {}

void payFees() override {

hasPaidFees = true;

cout<<name<<" has paid monthly transport fee."<<endl;

}

void tapCard() const override {

cout<<name<<" Teacher attendance recorded."<<endl;

}

};

class StaffMember : public User {

public:

StaffMember(string n, int id, string stop) : User(n, id, stop) {}

void payFees() override {

hasPaidFees = true;

cout<<name<<" has paid monthly transport fee."<<endl;

}

void tapCard() const override {

cout<<name<<" staff attendance recorded."<<endl;

}

};

class BusRoute {

private:

int routeID;

string routeName;

string\* stops;

int stopCount;

public:

BusRoute(int id, string name, string stopList[], int count) : routeID(id), routeName(name), stopCount(count) {

stops = new string[count];

for (int i = 0; i < count; i++) stops[i] = stopList[i];

}

void displayRoute() const {

cout<<"Route ID: "<<routeID<<endl<<"Name: "<<routeName<<endl<<"Stops: "<<endl;

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

cout<<stops[i] << " ";

cout<<endl;

}

}

bool operator==(const BusRoute& other) const {

return (routeID==other.routeID && routeName==other.routeName);

}

~BusRoute() { delete[] stops; }

};

class TransportSystem {

private:

User\*\* users;

BusRoute\*\* routes;

int userCount, routeCount;

int maxUsers, maxRoutes;

public:

TransportSystem(int userLimit, int routeLimit) : userCount(0), routeCount(0), maxUsers(userLimit), maxRoutes(routeLimit) {

users = new User\*[maxUsers];

routes = new BusRoute\*[maxRoutes];

}

void registerUser(User\* user) {

if (userCount < maxUsers) {

users[userCount++] = user;

}

else {

cout << "User limit reached!" << endl;

}

}

void addRoute(BusRoute\* route) {

if (routeCount < maxRoutes) {

routes[routeCount++] = route;

}

else {

cout << "Route limit reached!" << endl;

}

}

void displayUsers() const {

cout << "Registered Users:" << endl;

for (int i = 0; i < userCount; i++) users[i]->displayInfo();

}

void displayRoutes() const {

cout << "Available Routes:" << endl;

for (int i = 0; i < routeCount; i++) routes[i]->displayRoute();

}

~TransportSystem() {

for (int i = 0; i < userCount; i++) delete users[i];

for (int i = 0; i < routeCount; i++) delete routes[i];

delete[] users;

delete[] routes;

}

};

int main() {

cout<<"WAMIZA NOMAN 24K-0010\n"<<endl;

TransportSystem system(10, 10);

cout<<"FAST Transportation System\n"<<endl;

system.registerUser(new Student("Alaya", 53, "A"));

system.registerUser(new Teacher("Sir Ahmed", 201, "B"));

system.registerUser(new StaffMember("Sir Ali", 301, "C"));

string stops1[] = {"A", "B", "C"};

string stops2[] = {"D", "E", "F"};

BusRoute\* route1 = new BusRoute(1, "Route 1", stops1, 3);

BusRoute\* route2 = new BusRoute(2, "Route 2", stops2, 3);

system.addRoute(route1);

system.addRoute(route2);

system.displayUsers();

system.displayRoutes();

if (\*route1 == \*route2) {

cout<<"Same Routes"<<endl;

}

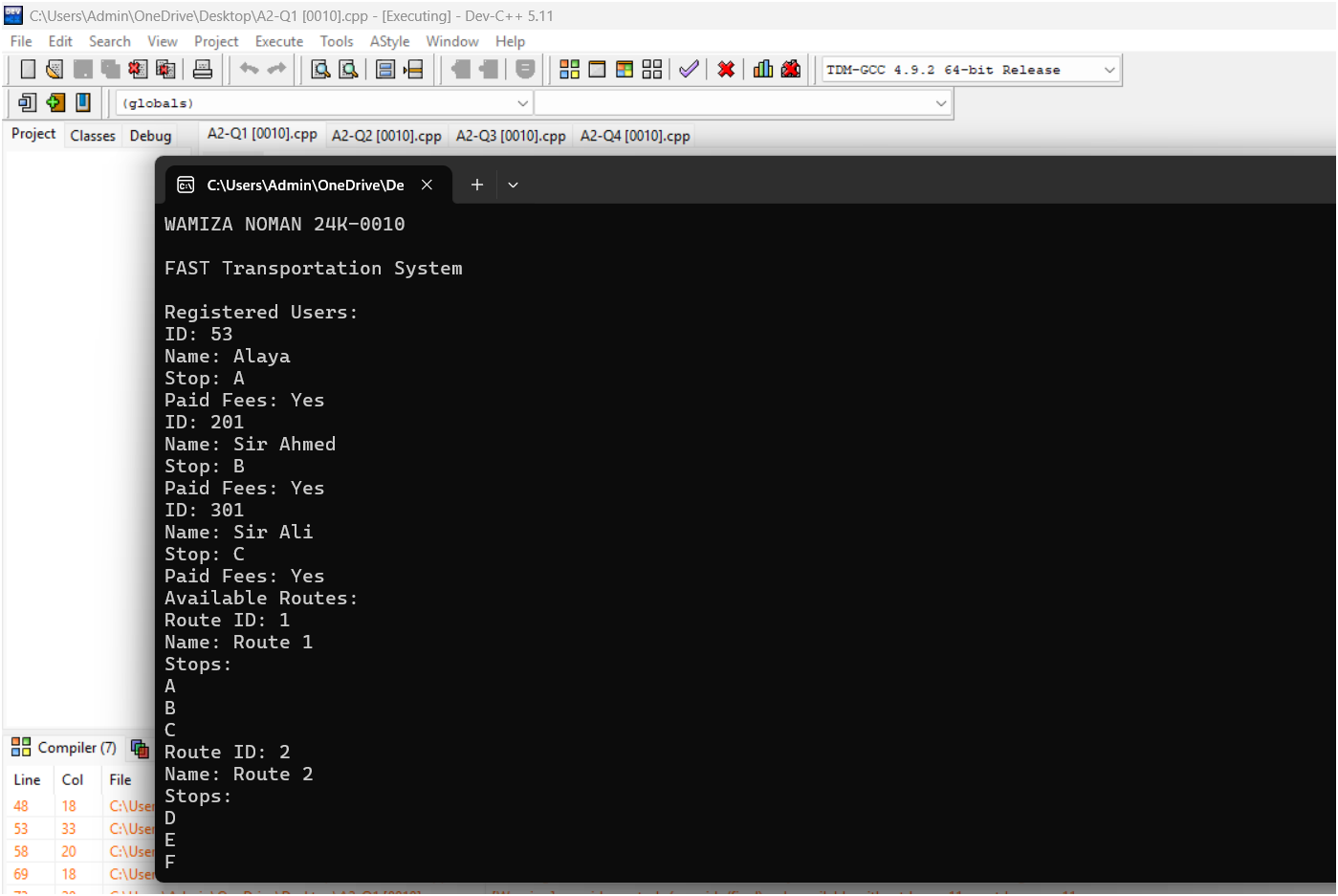
else {

cout<<"Routes are different."<<endl;

}

return 0;

}



Question 2:

Let’s say you are working to simulate a Haunted House based game! Your game allows users to create multiple Haunted Houses with ghosts of their own choices. After creating a Haunted House, the user can run a simulation to see how well the Haunted House is doing in terms or visitor experience and scares!

Let’s have a look at the entities involved:

• HauntedHouse – has a name and multiple ghosts (ensure runtime polymorphism here).

• Ghost – the super class for all types of ghosts. It will contain everything that is shared between different types of ghosts. For example: name of the worker playing the ghost, the scare level of the ghost (RNG 1-10), a function to perform the haunting, etc. Ghosts have some operator overloading done on the << operator to display information about them, and + operator to upgrade a ghost (results in a ghost with two people playing a bigger ghost and combined scare level of both ghosts)

• There are different ghosts who perform hauntings differently; Poltergeists (move objects), Banshees (scream loudly), and ShadowGhosts (whisper creepily).

• There are also some hybrid ghosts that may appear; for example, ShadowPoltergeist (is a shadow ghost and Poltergeist). This ghost performs both actions of a ShadowGhost and a Poltergeist.

• Visitor – a class of objects that can “visit” a Haunted house. Each visitor has a name and some amount of bravery (1-4: Cowardly, 5-7: Average, 8-10: Fearless). When a visitor visits a Haunted House, the ghosts will try to haunt them. If the scare level of the ghost is less than the bravery range of the visitor, then the visitor might laugh or taunt the ghost. If it is greater, then the visitor might scream or run away. If it’s around average, then they might get a shaky voice. (please make sure to check if the scare level lies within the bravery range; DO NOT compare the numbers directly)

• Main method should make 2-3 Haunted Houses with multiple ghosts of each type.

• Main method should also have an array of Visitors (a group of friends visiting the haunted house) Try to ensure that there’s at least one of each bravery level among the group.

• “Visit” is a global friend function that accepts the Visitors array and a Haunted House, and then simulates the visit.

• Do the same for each Haunted house that you made.

#include <iostream>

#include <string>

using namespace std;

// 24K-0010 WAMIZA NOMAN

class Ghost {

protected:

string workerName;

int scareLevel;

public:

Ghost(string name, int level) : workerName(name), scareLevel(level) {}

virtual void haunt() const = 0;

virtual Ghost\* clone() const = 0;

int getScareLevel() const {

return scareLevel;

}

friend ostream& operator<<(ostream& os, const Ghost& g) {

os << "Ghost Actor: " << g.workerName << ", Scare Level: " << g.scareLevel;

return os;

}

virtual Ghost\* operator+(const Ghost& other) const;

virtual ~Ghost() {}

};

class Poltergeist : public Ghost {

public:

Poltergeist(string name, int level) : Ghost(name, level) {}

void haunt() const override {

cout << workerName << " throws objects" << endl;

}

Ghost\* clone() const override {

return new Poltergeist(\*this);

}

};

class Banshee : public Ghost {

public:

Banshee(string name, int level) : Ghost(name, level) {}

void haunt() const override {

cout << workerName << " screams loudly!" << endl;

}

Ghost\* clone() const override {

return new Banshee(\*this);

}

};

class ShadowGhost : public Ghost {

public:

ShadowGhost(string name, int level) : Ghost(name, level) {}

void haunt() const override {

cout << workerName << " whispers creepily" << endl;

}

Ghost\* clone() const override {

return new ShadowGhost(\*this);

}

};

class ShadowPoltergeist : public Ghost {

public:

ShadowPoltergeist(string name, int level) : Ghost(name, level) {}

void haunt() const override {

cout << workerName << " throws objects and whispers creepily!" << endl;

}

Ghost\* clone() const override {

return new ShadowPoltergeist(\*this);

}

};

Ghost\* Ghost::operator+(const Ghost& other) const {

return new Poltergeist(workerName + "+" + other.workerName, scareLevel + other.scareLevel);

}

class Visitor {

public:

string name;

int bravery;

Visitor(string n, int b) : name(n), bravery(b) {}

void react(int ghostScareLevel) const {

if (bravery <= 4 && ghostScareLevel > bravery) {

cout << name << " runs away while screaming!" << endl;

}

else if (bravery >= 8 && ghostScareLevel < bravery) {

cout << name << " laughs and mocks the ghost!" << endl;

}

else {

cout << name << " shakes with a nervous voice..." << endl;

}

}

};

class HauntedHouse {

private:

string name;

Ghost\* ghosts[10];

int ghostCount;

public:

HauntedHouse(string n) : name(n), ghostCount(0) {}

void addGhost(Ghost\* g) {

if (ghostCount < 10)

ghosts[ghostCount++] = g->clone();

}

void display() const {

cout << "Haunted House: " << name << endl;

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

cout << \*ghosts[i] << endl;

}

}

void simulate(Visitor visitors[], int numVisitors) const {

cout << "\nSimulation for Haunted House: " << name << endl;

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

cout << "\nVisitor: " << visitors[i].name << endl;

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

ghosts[j]->haunt();

visitors[i].react(ghosts[j]->getScareLevel());

}

}

}

~HauntedHouse() {

for (int i = 0; i < ghostCount; ++i)

delete ghosts[i];

}

};

void Visit(Visitor visitors[], int numVisitors, const HauntedHouse& house) {

house.simulate(visitors, numVisitors);

}

int main() {

Visitor visitors[] = {

Visitor("Ali", 3),

Visitor("Bilal", 6),

Visitor("Dua", 9)

};

cout << "WAMIZA NOMAN 24K-0010\n" << endl;

HauntedHouse house1("Old Manor");

house1.addGhost(new Poltergeist("Hamza", 5));

house1.addGhost(new Banshee("Sana", 7));

house1.addGhost(new ShadowGhost("Nashit", 4));

HauntedHouse house2("Whispering Halls");

house2.addGhost(new ShadowPoltergeist("Maria", 8));

house2.addGhost(new Banshee("Faraz", 6));

HauntedHouse house3("Creepy Cellar");

house3.addGhost(new ShadowGhost("Ayesha", 3));

house3.addGhost(new Poltergeist("Usman", 9));

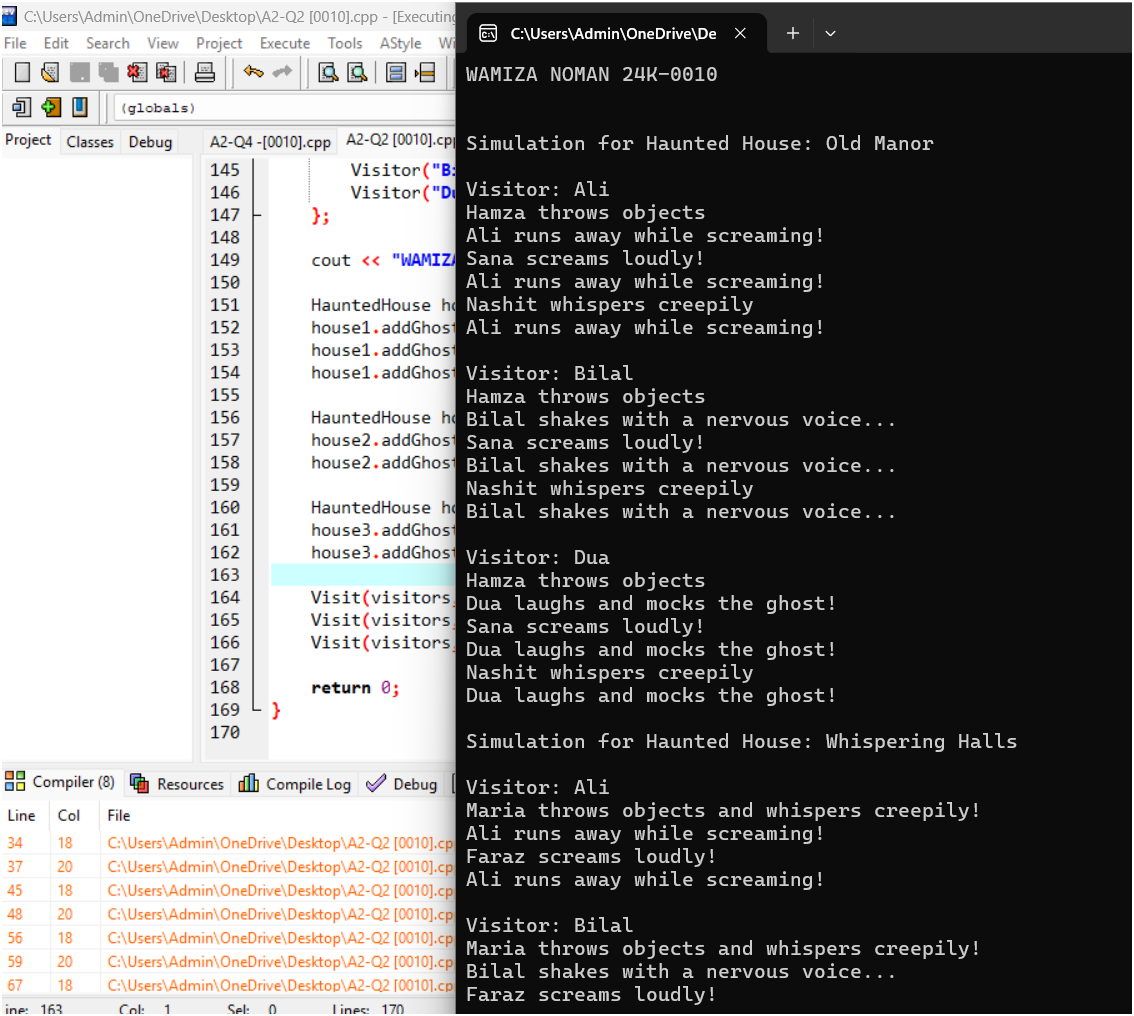
Visit(visitors, 3, house1);

Visit(visitors, 3, house2);

Visit(visitors, 3, house3);

return 0;

}



Question 3:

Ramzan Box Delivery System (Hints: Inheritance & Constructor Chaining, Function Overloading, Function Overriding & Polymorphism, Friend Function & Operator Overloading, Conflict Resolution via Friend Functions.) In the year 2030, a futuristic AI-driven delivery company, RamzanBox, specializes in delivering food and essential supplies during the holy month of Ramzan. The company operates a fleet of autonomous drones, time-traveling delivery ships, and high-speed ground pods to ensure that iftar meals and sehri packages reach their destinations on time. Each vehicle is equipped with an AI assistant that makes decisions based on package weight, urgency (Sehri/Iftar timing), and distance while ensuring efficient and timely deliveries. The system prevents scheduling conflicts and ensures that no package arrives late, especially when delivering food for fasting individuals. System Overview

• The Vehicle class represents all delivery vehicles, including drones, time ships, and high- speed hyperloop pods.

• A RamzanTimeShip is a special vehicle that can travel through time, ensuring that historical deliveries follow strict accuracy protocols.

• A RamzanDrone is an AI-powered aerial delivery system that specializes in delivering small iftar meals at high speed.

• A RamzanHyperPod is a ground-based high-speed transport system optimized for bulk deliveries.

• Each vehicle's AI assistant evaluates package priority based on urgency (Sehri or Iftar), weight, and destination.

• Vehicles operate on a shared AI network, but if two AI systems disagree on priority, a conflict resolution system determines the best course of action.

• Operators use customized control panels to issue delivery commands, but different vehicles interpret the same command based on their unique design and functionality.

• Vehicles may also compare efficiency metrics, especially when deciding which vehicle is best suited for critical food deliveries during Ramzan.

Class Structure & Functionality

1. Base Vehicle Class All delivery vehicles derive from a common base class that provides fundamental functionality. It includes:

• A unique vehicle ID

• A method to calculate the optimal delivery route

• A function to determine estimated delivery time

• A static variable to track the total number of active deliveries

2. Specialized Vehicles

• RamzanDrone: Small, fast, and airborne, designed for iftar meal deliveries.

• RamzanTimeShip: Ensures historical accuracy when delivering food to different time periods.

• RamzanHyperPod: High-speed underground transport optimized for bulk food deliveries.

Each vehicle type handles deliveries differently, even when given the same destination.

3. AI Decision-Making System

Each vehicle follows a different movement strategy based on its mode of operation:

• A RamzanDrone calculates an aerial route for high-speed delivery.

• A RamzanTimeShip verifies historical consistency before proceeding.

• A RamzanHyperPod navigates an underground tunnel network for efficient bulk delivery.

• All vehicles override a general movement function to ensure their unique operational logic.

4. Operator Control Panel

The RamzanBox AI system allows operators to issue delivery commands using a control panel:

command("Deliver", packageID);

command("Deliver", packageID, urgencyLevel);

• A RamzanTimeShip treats “urgent” packages as historically sensitive, requiring validation before transport.

• A RamzanDrone interprets “urgent” as activating high-speed mode to ensure an iftar meal arrives on time.

Even though the function name remains the same, different vehicle types interpret the command differently.

5. AI Conflict Resolution System In cases where two AI systems disagree on which vehicle should handle a delivery, a neutral decision-making system determines the best option.

• A friend function allows access to private attributes to compare vehicles based onefficiency.

• The == operator is overloaded so that vehicles can be directly compared based on speed, capacity, and energy efficiency.

#include <iostream>

#include <string>

using namespace std;

// 24K-0010 WAMIZA NOMAN

class vehicle {

protected:

string vehicleid;

float speed;

float capacity;

float energyeffeciency;

static int totaldeliveries;

public:

vehicle(string id, float sp, float cap, float eff) : vehicleid(id), speed(sp), capacity(cap), energyeffeciency(eff) {

totaldeliveries++;

}

virtual void move() {

cout << "vehicle " << vehicleid << " is moving generically."<< endl;

}

virtual void command(string action, int packageid) {

cout << "Base vehicle processing command: " << action << " for package " << packageid << "\n";

}

virtual void command(string action, int packageid, string urgency) {

cout << "Base vehicle processing command with urgency: " << action << ", Package: " << packageid << ", Urgency: " << urgency << "\n";

}

virtual float calculateETA(float distance) {

return distance / speed;

}

static int gettotaldeliveries() {

return totaldeliveries;

}

friend bool operator==(const vehicle& v1, const vehicle& v2);

friend vehicle resolveconflict(const vehicle& v1, const vehicle& v2);

};

int vehicle::totaldeliveries = 0;

bool operator==(const vehicle& v1, const vehicle& v2) {

return (v1.speed == v2.speed && v1.capacity == v2.capacity && v1.energyeffeciency == v2.energyeffeciency);

}

vehicle resolveconflict(const vehicle& v1, const vehicle& v2) {

if ((v1.speed + v1.energyeffeciency) > (v2.speed + v2.energyeffeciency))

{

return v1;

}

else

{

return v2;

}

}

class ramzandrone : public vehicle {

public:

ramzandrone(string id) : vehicle(id, 150.0, 5.0, 90.0) {}

void move() override {

cout<< "Drone " << vehicleid << " has taken flight and is navigating through the sky." << endl;

}

void command(string action, int packageid) override {

cout<< "Drone " << vehicleid << " delivering package " << packageid << " at normal speed."<< endl;

}

void command(string action, int packageid, string urgency) override {

if (urgency == "urgent")

cout<< "Drone " << vehicleid << " delivering URGENT Iftar package " << packageid << " at high speed."<< endl;

else

command(action, packageid);

}

};

class ramzantimeship : public vehicle {

public:

ramzantimeship(string id) : vehicle(id, 100.0, 10.0, 80.0) {}

void move() override {

cout<< "Timeship " << vehicleid << " verifying historical consistency and time traveling." << endl;

}

void command(string action, int packageid) override {

cout<< "Timeship " << vehicleid << " delivering package " << packageid << " in timeline." << endl;

}

void command(string action, int packageid, string urgency) override {

if (urgency == "urgent")

cout<< "TimeShip " << vehicleid << " is validating sensitive of URGENT package " << packageid << " within historical context.\n";

else

command(action, packageid);

}

};

class ramzanhyperpod : public vehicle {

public:

ramzanhyperpod(string id) : vehicle(id, 120.0, 50.0, 85.0) {}

void move() override {

cout<< "HyperPod " << vehicleid << " navigating underground tunnel network." << endl;

}

void command(string action, int packageid) override {

cout<< "HyperPod " << vehicleid << " transporting bulk package " << packageid << ".\n";

}

void command(string action, int packageid, string urgency) override {

if (urgency == "urgent")

cout<< "HyperPod " << vehicleid << " handling URGENT bulk delivery for package " << packageid << ".\n";

else

command(action, packageid);

}

};

int main() {

cout<<"WAMIZA NOMAN 24K-0010\n"<<endl;

ramzandrone drone1("DRN0017");

ramzantimeship ship1("TS003");

ramzanhyperpod pod1("HYP055");

drone1.move();

ship1.move();

pod1.move();

drone1.command("Deliver", 301);

ship1.command("Deliver", 502, "urgent");

pod1.command("Deliver", 303, "normal");

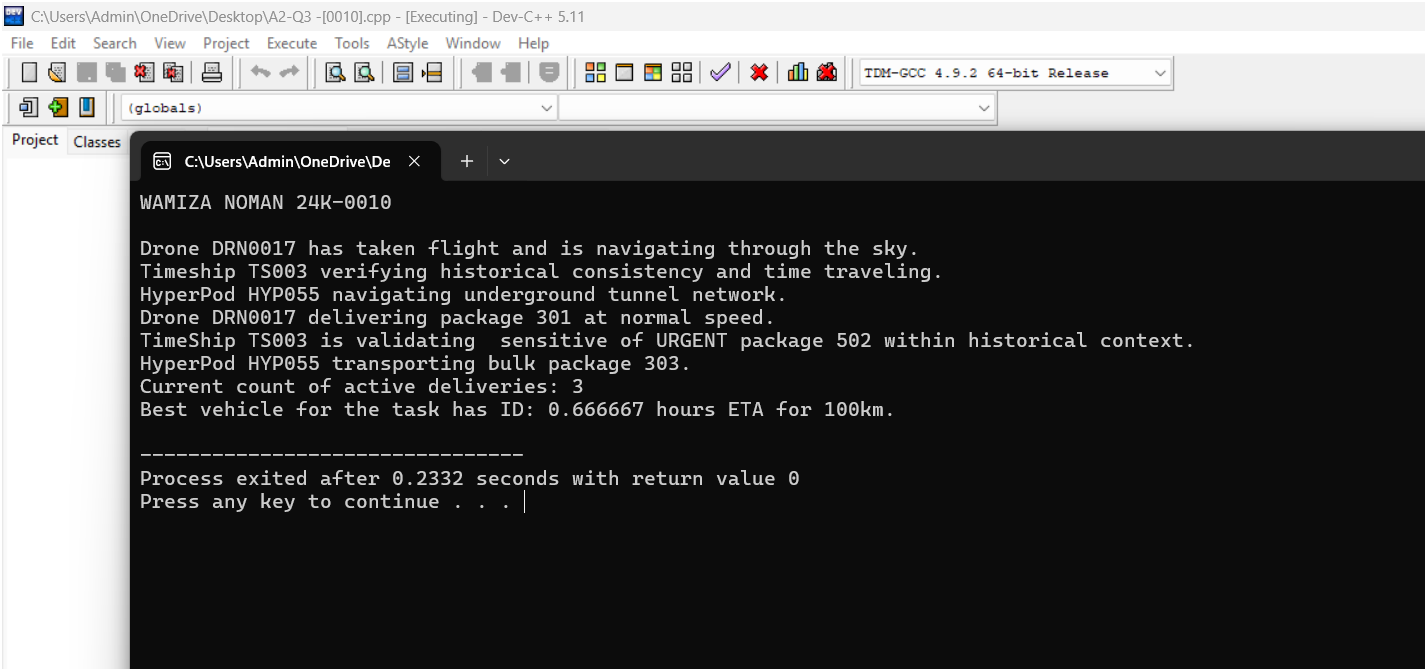
cout<< "Current count of active deliveries: " << vehicle::gettotaldeliveries() << "\n";

vehicle best = resolveconflict(drone1, pod1);

cout<< "Best vehicle for the task has ID: " << best.calculateETA(100.0) << " hours ETA for 100km.\n";

return 0;

}



Question 4:

A university has a Lab Management System used by undergraduate students. The system manages different types of users, such as Students, Teaching Assistants (TAs), and Professors. Students can access basic tools and complete assignments. TAs monitor students, assist in labs. Professors have full control of labs, assign projects, and manage research.PERMISSIONS: STUDENT = submit assignment , TA = view projects, manage\_students, PROFESSOR = assign projects , full\_lab\_access. Create a global function that generates the hashes of the password. Use the value given to calculate the hash. [ hash = 5381]

Then use the following formula to calculate the hash [ hash \* 33 + c ] where “c” represents a single character. Create a global function authenticateAndPerformAction(User\* user, string action) that performs the functionality based on the User roles and their permissions.

The User class:

• Has basic attributes like name, ID, list of permissions, email and hashed\_password.

• A parameterized constructor to set the attributes and save passwords as hashed.

• An authenticate function which authenticates based on the correct password.

• Display function to display the information.

• A function access lab that checks the permissions of the user and provides access to the lab if allowed.

The Student class:

• Derived from the user class

• Override the display function

• Has a list of assignments where each index represents 1 assignment. If the assignment is submitted by the student update the status to 1 or else leave it at 0.

• Has a functionality where students can be given assignments and their status can be updated after submission.

The TA class:

• Derived from the Student class

• Override the display function

• Has a List of Students that are assigned to it.

• Has a List of projects that they are working on with the professor (MAX = 2) and provides functionality to view the current projects or start working on a new project if the limit is not exceeded.

• Has a functionality where the permission is checked and then students can be assigned to the TA. Each TA can manage only up to 10 students. The TA cannot be assigned more than 10 at any time.

The Professor class:

• Derived from the user class

• Override the display function

• Has a functionality that allows professors to work with TA’s on projects. At any one time each TA can only work on 2 projects.

Create objects of all the classes. Call the display function and authenticate function as needed.

#include <iostream>

#include <string>

using namespace std;

//24k-0010 WAMIZA NOMAN

long calculateHash(string password) {

long hash = 5381;

for (char c : password) {

hash = hash \* 33 + c;

}

return hash;

}

class User {

protected:

string name;

string id;

string email;

long hashedPassword;

string permissions[3];

public:

User(string name, string id, string permissions[], int permSize, string email, string password) {

this->name = name;

this->id = id;

this->email = email;

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

this->permissions[i] = permissions[i];

}

this->hashedPassword = calculateHash(password);

}

string getName() const {

return name;

}

bool authenticate(string password) {

return this->hashedPassword == calculateHash(password);

}

virtual void display() {

cout << "User Information:" << endl;

cout << "Name: " << name << endl;

cout << "ID: " << id << endl;

cout << "Email: " << email << endl;

cout << "Permissions: ";

for (const string &permission : permissions) {

cout << permission << " ";

}

cout << endl;

}

bool accessLab(string action) {

for (const string &permission : permissions) {

if (permission == action) {

cout << "Access granted: " << name << " can perform " << action << endl;

return true;

}

}

cout << "Access denied: " << name << " doesn't have permission to " << action << endl;

return false;

}

};

class Student : public User {

private:

int assignments[10];

int assignmentCount;

public:

Student(string name, string id, string email, string password) : User(name, id, new string[1]{"submit\_assignment"}, 1, email, password), assignmentCount(0) {}

void display() override {

cout << "--- Student Information ---" << endl;

User::display();

cout << "Assignments Status: ";

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

cout << "Assignment " << (i + 1) << ": "

<< (assignments[i] == 1 ? "Submitted" : "Pending") << " | ";

}

cout << endl;

}

void addAssignment() {

if (assignmentCount < 10) {

assignments[assignmentCount++] = 0;

cout << "New assignment added. Total assignments: " << assignmentCount << endl;

} else {

cout << "Cannot add more assignments. Maximum limit reached." << endl;

}

}

void submitAssignment(int assignmentNumber) {

if (assignmentNumber > 0 && assignmentNumber <= assignmentCount) {

assignments[assignmentNumber - 1] = 1;

cout << "Assignment " << assignmentNumber << " submitted successfully." << endl;

} else {

cout << "Invalid assignment number." << endl;

}

}

};

class TA : public Student {

private:

Student\* students[10];

int studentCount;

string projects[2];

int projectCount;

public:

TA(string name, string id, string email, string password)

: Student(name, id, email, password), studentCount(0), projectCount(0) {

permissions[0] = "view\_projects";

permissions[1] = "manage\_students";

}

void display() override {

cout << "\n--- TA Information ---" << endl;

User::display();

cout << "Assigned Students: ";

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

cout << students[i]->getName() << " ";

}

cout << endl;

cout << "Projects: ";

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

cout << projects[i] << " ";

}

cout << endl;

}

bool addStudent(Student\* student) {

if (studentCount < 10) {

students[studentCount++] = student;

cout << student->getName() << " assigned to TA " << getName() << endl;

return true;

} else {

cout << "TA " << getName() << " cannot manage more than 10 students." << endl;

return false;

}

}

bool addProject(string projectName) {

if (projectCount < 2) {

projects[projectCount++] = projectName;

cout << "Project '" << projectName << "' added to TA " << getName() << endl;

return true;

} else {

cout << "TA " << getName() << " cannot manage more than 2 projects." << endl;

return false;

}

}

};

class Professor : public User {

public:

Professor(string name, string id, string email, string password)

: User(name, id, new string[2]{"assign\_projects", "full\_lab\_access"}, 2, email, password) {}

void display() override {

cout << "\n--- Professor Information ---" << endl;

User::display();

}

void assignProject(TA\* ta, string projectName) {

if (ta->addProject(projectName)) {

cout << "Project '" << projectName << "' assigned to TA " << ta->getName() << endl;

} else {

cout << "TA " << ta->getName() << " cannot take more projects." << endl;

}

}

};

void authenticateAndPerformAction(User\* user, string action) {

string password;

cout << "Enter password for " << user->getName() << ": ";

cin >> password;

if (user->authenticate(password)) {

cout << "Authentication successful." << endl;

user->accessLab(action);

} else {

cout << "Authentication failed." << endl;

}

}

int main() {

cout<<"WAMIZA NOMAN 24K-0010\n"<<endl;

cout<<"---------------------------------------"<<endl;

Student student("Wamiza Noman", "24k-0010", "k240010@nu.edu.pk", "password123");

student.addAssignment();

student.addAssignment();

TA ta("Talha Shahid", "23k-0465", "k230465@nu.edu.pk", "ta\_pass123");

ta.addStudent(&student);

ta.addProject("Project A");

Professor professor("Nahid Hanif", "P789", "nahid\_Hanif@nu.edu.pk", "prof\_pass123");

professor.assignProject(&ta, "Research Project");

student.submitAssignment(1);

cout << "\nDisplaying all users:" << endl;

student.display();

ta.display();

professor.display();

authenticateAndPerformAction(&student, "submit\_assignment");

authenticateAndPerformAction(&ta, "view\_projects");

authenticateAndPerformAction(&professor, "assign\_projects");

return 0;

