**Mehdi Miraç ARAT**

**2121251023**

**Software Engineering**

**Project Documentation: Flower Sales Management System**

**FLOWERSALES\_IMP.CPP**

1. Introduction: The Flower Shop Management System is a C++ program designed to help manage the inventory of a flower shop. It allows users to add, update, delete, and search for flower data, as well as place flower orders. Additionally, it provides user authentication and file handling for storing flower and user data.
2. Program Components: The program consists of several components:
   * **Flower** class: Represents individual flowers with attributes like ID, name, price, type, and stock.
   * **FlowerType** enum: Defines the types of flowers available (e.g., Rose, Tulip, Lily).
   * **UserBase** class: Represents user information with attributes like username, password, and admin status.
   * **admin** and **User** classes: Inherited classes from **UserBase** to differentiate between admin and standard users.
3. Data Structures:
   * **flowerDatabase**: An array of **Flower** objects to store flower data.
   * **flowerCount**: An integer to keep track of the number of flowers in the database.
   * **flowerorderdb**: A dynamic array of **Flower** objects to store flower orders.
   * **flowercount**: An integer to keep track of the number of flower orders.
4. Functions:
   * **initializeFlowerDatabase()**: Initializes the flower database by setting **flowerCount** to zero.
   * **initializeFlowerOrderDatabase()**: Initializes the flower order database by setting **flowercount** to zero.
   * **loadFlowerDataFromFile()**: Reads flower data from a file and populates **flowerDatabase**.
   * **loadFlowerorderDataFromFile()**: Reads flower order data from a file and populates **flowerorderdb**.
   * **writeFlowerDataToFile()**: Writes flower data to a file.
   * **writeFlowerOrdersToFile()**: Writes flower orders to a file.
   * **deleteFlowerFromFile()**: Deletes a flower from the file by its ID.
   * **addFlower()**: Adds a new flower to the database.
   * **findFlower()**: Searches for a flower by its ID and displays its details.
   * **updateFlower()**: Updates the information of a flower by its ID.
   * **deleteFlower()**: Deletes a flower from the database by its ID.
   * **summarizeFlowerSales()**: Calculates and displays the total sales of all flowers.
   * **userLogin()**: Authenticates users based on provided username and password.
   * **initializeUsers()**, **initializeadmins()**, **initializestandardUsers()**: Loads user data from a file and initializes user structures.
   * **writeUsersToFile()**: Writes user data to a file.
5. User Authentication:
   * The program uses a user authentication system to distinguish between admin and standard users.
   * Admin users have access to all functions, while standard users have limited access.
6. File Handling:
   * Flower and user data is read from and written to text files (**flowers.txt** and **users.txt**).
   * The program handles file opening, reading, and writing errors.
7. Usage:
   * Users can interact with the program through a command-line interface.
   * They can perform operations like adding, updating, and deleting flowers, searching for flowers, placing flower orders, and viewing sales summaries.
   * Admin users can also manage user accounts.
8. Error Handling:
   * The program includes error checks for file operations and user inputs.
   * It provides informative error messages to guide users.
9. Conclusion: The Flower Shop Management System is a versatile C++ program designed for efficiently managing flower shop operations. It offers the ability to handle flower data, orders, and user accounts, making it a useful tool for flower shop owners and employees. Users can easily customize and extend its functionality to suit their specific needs.

**FLOWESSALSE.HPP**

1. Program Components: The program consists of several components:
   * **Flower** class: Represents individual flowers with attributes like ID, name, price, type, and stock.
   * **FlowerType** enum: Defines the types of flowers available (e.g., Rose, Tulip, Lily).
   * **UserBase** class: Represents user information with attributes like username, password, and admin status.
   * **admin** and **User** classes: Inherited classes from **UserBase** to differentiate between admin and standard users.
2. Data Structures:
   * **flowerDatabase**: An array of **Flower** objects to store flower data.
   * **flowerCount**: An integer to keep track of the number of flowers in the database.
   * **flowerorderdb**: A dynamic array of **Flower** objects to store flower orders.
   * **flowercount**: An integer to keep track of the number of flower orders.
3. Functions:
   * **initializeFlowerDatabase()**: Initializes the flower database by setting **flowerCount** to zero.
   * **initializeFlowerOrderDatabase()**: Initializes the flower order database by setting **flowercount** to zero.
   * **loadFlowerDataFromFile()**: Reads flower data from a file and populates **flowerDatabase**.
   * **loadFlowerorderDataFromFile()**: Reads flower order data from a file and populates **flowerorderdb**.
   * **writeFlowerDataToFile()**: Writes flower data to a file.
   * **writeFlowerOrdersToFile()**: Writes flower orders to a file.
   * **deleteFlowerFromFile()**: Deletes a flower from the file by its ID.
   * **addFlower()**: Adds a new flower to the database.
   * **findFlower()**: Searches for a flower by its ID and displays its details.
   * **updateFlower()**: Updates the information of a flower by its ID.
   * **deleteFlower()**: Deletes a flower from the database by its ID.
   * **summarizeFlowerSales()**: Calculates and displays the total sales of all flowers.
   * **userLogin()**: Authenticates users based on provided username and password.
   * **initializeUsers()**, **initializeadmins()**, **initializestandardUsers()**: Loads user data from a file and initializes user structures.
   * **writeUsersToFile()**: Writes user data to a file.
4. User Authentication:
   * The program uses a user authentication system to distinguish between admin and standard users.
   * Admin users have access to all functions, while standard users have limited access.
5. File Handling:
   * Flower and user data is read from and written to text files (**flowers.txt** and **users.txt**).
   * The program handles file opening, reading, and writing errors.
6. Usage:
   * Users can interact with the program through a command-line interface.
   * They can perform operations like adding, updating, and deleting flowers, searching for flowers, placing flower orders, and viewing sales summaries.
   * Admin users can also manage user accounts.
7. Error Handling:
   * The program includes error checks for file operations and user inputs.
   * It provides informative error messages to guide users.

**Main Function Documentation**

The **main** function serves as the entry point for the Flower Sales System program. It performs several key tasks, including initializing data, handling user authentication, and providing options for user interaction.

1. **Header Includes**:
   * **#include <iostream>**: Provides input and output stream functionality.
   * **#include <fstream>**: Allows file input and output operations.
   * **#include <string>**: Enables string manipulation.
   * **#include <vector>**: Utilizes dynamic arrays for storing user and manager data.
   * **"flowersales.hpp"**: Includes the header file for the Flower Sales System classes and enums.
   * **"flowersales\_imp.cpp"**: Includes the implementation for functions related to flower operations.
   * **"messagesandmanager.cpp"**: Includes the implementation for functions related to messages and manager operations.
2. **Initialization**:
   * **initializeFlowerDatabase()**: Initializes the flower database.
   * **initializeFlowerOrderDatabase()**: Initializes the flower order database.
   * **loadFlowerDataFromFile()**: Loads flower data from a file.
   * **loadFlowerorderDataFromFile()**: Loads flower order data from a file.
3. **User and Manager Vectors**:
   * Several vectors are declared to dynamically store user, manager, standard user, admin, and register message data.
4. **Reading Data from Files**:
   * User and admin data are initialized from files using functions like **readmanagersfromfile**, **initializeUsers**, and **initializeadmins**.
5. **Default Admin User**:
   * If there are no user accounts, a default admin account is created and stored in the **users** vector with the username "admin" and password "admin123."
6. **Authentication Loop**:
   * A loop allows users to select login options, manager login, or registration.
   * User inputs are validated for correctness and within a valid range.
7. **Login**:
   * For normal users, the program requests a username and password for authentication.
   * If successful, the user is marked as logged in (**isLoggedIn = true**).
8. **Manager Panel**:
   * If the user is an admin, they gain access to the manager panel.
   * The manager panel provides options for managing flowers, orders, users, and messages.
   * Functions such as **adduser**, **deleteUser**, **placeOrder**, and more are available.
9. **User Panel**:
   * If the user is not an admin, they gain access to the user panel.
   * The user panel provides options for placing orders, listing flowers, and finding flowers by ID.
10. **Exiting**:
    * Users can choose to exit the program at any time, which results in the program terminating and returning to the main menu.
11. **Error Handling**:
    * The program includes error handling for invalid input and provides appropriate error messages to the user.
12. **Return**:
    * The **main** function returns 0 upon successful execution.

**MESSAGEANDMANAGER.CPP**

Introduction

This C++ program provides a simple message management system, where users (Managers) can write, list, and manage messages. It includes features like reading/writing data from/to files and uses object-oriented programming concepts such as polymorphism and dynamic memory allocation.

Code Structure

The program consists of several components:

1. Message Base Struct (Message): This struct serves as the base class for different message types. It contains a messageText field and a virtual function getType() to facilitate polymorphism.
2. Manager Struct (Manager): This struct represents a manager user. It contains username, password, an array of pointers to messages (messages), and a messageCount field to keep track of the number of messages. Managers can write messages, list messages, read messages, and manage their data.
   * writeMessage(const std::string& text): Allows a manager to write and store a new message.
   * listMessages(): Lists all the messages associated with the manager.
   * readmessages(): Reads and displays messages.
   * friend void showtotalarr(const Manager& manager): A friend function that shows the total message count for a manager.
3. RegisterMessage Struct (RegisterMessage): This is a specialized message type that inherits from the Message base class. It includes a type field and overrides the getType() function to provide customized message types.
4. File Handling Functions:
   * readmanagersfromfile(std::vector<Manager>& managers, const std::string& filename): Reads manager data (username and password) from a file and populates the managers vector.
   * writemanagerstofile(const std::vector<Manager>& managers, const std::string& filename): Writes manager data to a file.
   * addToMessageArray(const std::string& messageType, std::vector<RegisterMessage>& messageArray): Adds a new message of a specified type to a vector of messages.
   * writemessagestotxt(std::vector<RegisterMessage>& messagearray): Writes message data to a text file.
   * readmessagesfromtxt(std::vector<RegisterMessage>& messageArray): Reads message data from a text file and populates the messageArray.
   * readregistermessages(std::vector<RegisterMessage>& messageArray): Reads and displays messages from the messageArray.
   * initializeregistremessagesfromtxt(std::vector<RegisterMessage>& messageArray): Initializes the messageArray by reading message data from a text file.
5. Manager Management Functions:
   * addmanager(std::vector<Manager>& managers, const std::string& username, const std.::string& password): Adds a new manager to the managers vector.
   * deletemanager(std::vector<Manager>& managers, const std::string& username): Deletes a manager from the managers vector.
   * listmanagers(const std::vector<Manager>& managers): Lists all the manager usernames.
   * updatemanager(std::vector<Manager>& managers, const std::string& username, const std::string& password): Updates the password of an existing manager.
   * findmanager(const std::vector<Manager>& managers, const std::string& username): Searches for a manager by username and displays their information.

Usage

1. Writing and Managing Messages:
   * Create a manager using the Manager struct.
   * Use writeMessage(const std::string& text) to write and store messages.
   * Use listMessages() to list all messages.
   * Use readmessages() to read and display messages.
2. Reading/Writing Managers and Messages to/from Files:
   * Use readmanagersfromfile to read manager data from a file into a vector of managers.
   * Use writemanagerstofile to write manager data to a file.
   * Use addToMessageArray to add messages to a vector of messages.
   * Use writemessagestotxt to write message data to a text file.
   * Use readmessagesfromtxt to read message data from a text file.
   * Use readregistermessages to read and display messages from the messageArray.
   * Use initializeregistremessagesfromtxt to initialize the messageArray from a text file.
3. Manager Management:
   * Use addmanager to add a new manager to the system.
   * Use deletemanager to delete a manager from the system.
   * Use listmanagers to list all the manager usernames.
   * Use updatemanager to update a manager's password.
   * Use findmanager to search for a manager by username and display their information.

The main function orchestrates the Flower Sales System, allowing users and managers to interact with the system, place orders, manage flowers and users, and send/receive messages. It provides a user-friendly interface with input validation and error handling for a smooth user experience.

**UML**

metin, diyagram, plan, paralel içeren bir resim

Açıklama otomatik olarak oluşturuldu