**Total marks: 7.5**

**—————**

**Object Oriented Programming Techniques**

**Final Project**

**Loot Lo: Console-Based E-commerce Store**

**Submitted to: Saira Shaheen**

| **Submitted By: Group Members** | | |
| --- | --- | --- |
| **Name** | **Reg.No.** | **Marks** |
| **Abdul Ghaffar** | **2312302** |  |
| **Hassan Raza** | **2312315** |  |
| **Ghulam Abbas** | **2312395** |  |
| **Wasi Ur Rehman Turk** | **2312345** |  |
| **Hamza Najib** | **2312312** |  |

# **Loot Lo: Console-Based E-commerce Store**

## **Introduction**

Loot Lo is a console-based e-commerce application designed to provide a seamless shopping experience for users. It offers a variety of functionalities, including user management, product browsing, cart handling, and order placement, all through a text-based user interface. The system is implemented using object-oriented programming principles and is structured to ensure modularity, scalability, and ease of maintenance.

## **Key Features**

### **1. User Management**

Loot Lo supports two types of users:

* **Admins**: Manage products, view sales reports, and oversee the system.
* **Clients**: Browse products, add items to their cart, and place orders.

### **2. Product Management**

* Products have unique IDs, names, prices, quantities, and descriptions.
* Admins can add, remove, or update product details.
* Clients can view the list of available products.

### **3. Cart System**

* Clients can add products to their cart.
* Remove or modify quantities of items in the cart.
* View the total price of their cart items.

### **4. Order Management**

* Clients can place orders, which are recorded in their order history.
* Admins can view all orders and track sales statistics.

### **5. Persistent Storage**

* User data, product inventory, and cart information are stored in text files.
* Ensures data is saved even after the application is closed.

## 

## **Application Structure**

### **1. Class Overview**

* **User Class**: Handles user details such as username, email, password, and order history.
* **Product Class**: Manages product attributes like ID, name, price, and description.
* **Cart Class**: Provides functionality for cart operations such as adding, removing, and calculating totals.
* **FileHandler Class**: Handles reading and writing to text files for persistent storage.
* **Client Class**: Implements the client-side operations, including browsing and purchasing products.

### **2. Modular Design**

Each functionality is encapsulated in a dedicated class, ensuring:

* Clean separation of concerns.
* Reusability of code components.
* Simplified debugging and enhancement.

## **Workflow**

### **1. Admin Panel**

* Admin logs in using their credentials.
* Adds, removes, or updates products.
* Views sales reports and system statistics.

### **2. Client Panel**

* Client logs in or registers an account.
* Browses the product catalog.
* Adds items to the cart and views cart details.
* Places an order and reviews their order history.

### **3. File Handling**

* All data is stored in structured text files within a files directory:
  + files/users: Stores user information.
  + files/products: Stores product inventory.
  + files/cart: Stores user-specific cart details.

## 

## **Example Usage**

### **Admin**

* **Add Product**: Add a new product with details like ID, Name, Price, Quantity, Description.
* **Update Product**: Modify the price or quantity of an existing product.
* **View Reports**: Get an overview of sales and user activity.

### **Client**

* **Browse Products**: View all available products with their details.
* **Manage Cart**: Add or remove items, adjust quantities, and calculate the total.
* **Place Orders**: Confirm purchases and review order history.

## **Implementation Highlights**

* **Object-Oriented Programming**: Core functionalities are built using OOP principles to ensure extensibility.
* **Dynamic Lists**: Custom List class for dynamic storage and management of products, users, and orders.
* **Error Handling**: Validates user input and ensures file operations are robust.
* **Custom Text Display**: Dynamic colored text output enhances user interaction.

## **Future Enhancements**

* Implement a graphical user interface (GUI) for a more intuitive experience.
* Add support for online payments and invoicing.
* Introduce advanced search and filtering options for products.
* Develop an API for mobile or web integrations.

## **Conclusion**

Loot Lo is a robust console-based e-commerce system that demonstrates effective use of object-oriented programming and file handling. With its modular design and extensive features, it serves as a foundational platform for developing full-scale e-commerce applications.

**Project Structure**

**Loot lo/**

|

|-- main.cpp # Entry point of the application

|-- authentication.cpp # Handles user authentication

|-- user.cpp # Manages user information and operations

|-- admin.cpp # Manages admin panel

|-- client.cpp # Manages client panel

|-- product\_manager.cpp # Manages products

|-- product.cpp # Defines product-related functionality

|-- cart.cpp # Manages shopping cart operations

|-- file\_handler.cpp # Handles file reading and writing

|-- list.cpp # Implements a custom list class for data structures

|-- text.cpp # Contains text animations

|

|-- files/ # Data storage folder

|

|-- cart/ # Contains user-specific cart files

| |-- <username>.txt # Cart data for each user

|

|-- product.txt # Store products information

|-- users.txt # Stores user credentials and details

**Source Code**

**main.cpp**

#include <iostream>

#include "text.cpp"

#include "list.cpp"

#include "cart.cpp"

#include "authentication.cpp"

#include "admin.cpp"

#include "client.cpp"

#include "product\_manager.cpp"

using namespace std;

// Takking username and password

List<string> takeUsernamePassword()

{

List<string> credentials;

string username, password;

// Taking user name

SingleColorText s("Enter username: ", Text::blue());

s.print();

cin >> username;

// Taking password

s.setText("Enter password: ");

s.print();

cin >> password;

credentials.push\_back(username);

credentials.push\_back(password);

return credentials;

}

int main()

{

// ----- Logo printing for the first time -----

// Logo colors

List<string> colors(10);

colors.push\_back(Text::red());

colors.push\_back(Text::green());

colors.push\_back(Text::blue());

// Logo Text

string logoText = ".\_\_\_\_ \_\_ .\_\_\_\_ \n"

"| | \_\_\_\_ \_\_\_\_\_/ |\_ | | \_\_\_\_ \n"

"| | / \_ \\ / \_ \\ \_\_\\ | | / \_ \\ \n"

"| |\_\_( <\_> | <\_> ) | | |\_\_( <\_> )\n"

"|\_\_\_\_\_\_\_ \\\_\_\_\_/ \\\_\_\_\_/|\_\_| |\_\_\_\_\_\_\_ \\\_\_\_\_/ \n"

" \\/ \\/ ";

MultiColorText logo = MultiColorText(logoText, colors, 20);

logo.print();

cout << endl

<< endl;

// Authencation handling if user goes to any protected screen

Authentication auth;

int choice;

do

{

// Printing menu

cout << endl

<< "1. Client panel"

<< endl

<< "2. Admin panel"

<< endl

<< "3. Create an account"

<< endl

<< "4. See all products"

<< endl

<< "0. Exit" << endl;

// creating object of animated text for input

SingleColorText s("Enter your choice: ", Text::blue(), false, 50);

s.print();

cin >> choice;

switch (choice)

{

case 1:

{

// Checking if already loggedin so don't take credentials

if (auth.isLoggedIn())

{

// const User \*user = User::findUserByUsername(auth.getCurrentUser()->getUsername());

Client client(auth.getCurrentUser());

client.begin();

}

else

{

while (true)

{

List<string> credentials = takeUsernamePassword();

// loggingIn

bool result = auth.login(credentials[0], credentials[1], "client");

if (result)

{

Client client(auth.getCurrentUser());

client.begin();

break;

}

else

{

s.setText("Error: Wrong credentials");

s.setColor(Text::red());

s.print();

cout << endl;

}

}

}

break;

}

case 2:

{

Admin admin;

if (auth.isLoggedIn() && auth.getCurrentUser()->getRole() == "admin")

{

admin.begin();

}

else

{

List<string> credentials = takeUsernamePassword();

bool result = auth.login(credentials[0], credentials[1], "admin");

if (result && auth.getCurrentUser()->getRole() == "admin")

{

admin.begin();

}

else

{

s.setText("Error: Wrong credentials");

s.setColor(Text::red());

s.print();

}

}

break;

}

case 3:

{

string username, password, email, phoneNumber, address, role = "client";

List<string> usernameAndPass = takeUsernamePassword();

username = usernameAndPass[0];

password = usernameAndPass[1];

s.setColor(Text::blue());

s.setText("Enter email: ");

s.print();

cin >> email;

s.setText("Enter phone numbmer: ");

s.print();

cin >> phoneNumber;

s.setText("Enter address: ");

s.print();

cin >> address;

bool result = auth.registerUser(username, password, email, role, phoneNumber, address);

if (result)

auth.login(username, password, "client");

// redirecting to the client panel instantly

Client client(User::findUserByUsername(auth.getCurrentUser()->getUsername(), auth.getAllUsers()));

client.begin();

break;

}

case 4:

{

ProductManager pm("files/product.txt");

pm.displayCatalog();

break;

}

case 0:

s.setText("Bye! have a good day.");

s.setColor(Text::green());

exit(0);

default:

{

s.setColor(Text::red());

s.setText("Invalid choice, Please try again.");

cout << endl;

s.print();

cout << endl;

}

cin.ignore();

}

} while (choice != 0);

cout << endl

<< endl;

return 0;

}

**authentication.cpp**  
#ifndef AUTH

#define AUTH

#include <iostream>

#include <string>

#include <fstream>

#include <sstream>

#include "list.cpp"

#include "user.cpp"

#include "file handler.cpp"

using namespace std;

class Authentication

{

private:

List<User> users; // Stores registered users

User \*loggedInUser; // Pointer to the currently logged-in user

string filename; // File to store user data

public:

// Constructor with filename for persistence

Authentication(const string &filename = "files/users.txt", int maxUsers = 100);

// Register and login functionalities

bool registerUser(const string &username, const string &password, const string &email, const string &role,

const string &phoneNumber = "", const string &address = "", const string &dateOfRegistration = "");

bool login(const string &username, const string &password, const string &role);

void logout();

// User management

bool isLoggedIn() const;

User \*getCurrentUser() const;

void displayAllUsers() const;

List<User> getAllUsers();

private:

// File handling methods

void loadUsersFromFile();

void appendUserToFile(const User &user);

// Helper function to find user by username

User \*findUserByUsername(const string &username);

};

// Constructor loads users from file

Authentication::Authentication(const string &filename, int maxUsers) : filename(filename), users(maxUsers), loggedInUser(nullptr)

{

loadUsersFromFile();

}

// Register a new user and append to file

bool Authentication::registerUser(const string &username, const string &password, const string &email, const string &role,

const string &phoneNumber, const string &address, const string &dateOfRegistration)

{

if (findUserByUsername(username) != nullptr)

{

cout << "Username already exists. Registration failed." << endl;

return false;

}

User newUser(username, password, email, role, phoneNumber, address, dateOfRegistration);

if (users.push\_back(newUser))

{

appendUserToFile(newUser); // Save new user to file

cout << "User registered successfully." << endl;

return true;

}

cout << "Registration failed. User list may be full." << endl;

return false;

}

// Login function

bool Authentication::login(const string &username, const string &password, const string &role)

{

User \*user = findUserByUsername(username);

// cout << "username" << user->getUsername() << endl

// << "password" << user->getPassword() << endl

// << "role" << user->getRole() << endl;

if (user && user->getPassword() == password && (user->getRole() == "admin" || user->getRole() == role))

{ // Check password

this->loggedInUser = user;

cout << "Login successful." << endl;

return true;

}

return false;

}

// Logout function

void Authentication::logout()

{

if (loggedInUser)

{

loggedInUser = nullptr;

cout << "Logged out successfully." << endl;

}

else

{

cout << "No user is currently logged in." << endl;

}

}

// Check if user is logged in

bool Authentication::isLoggedIn() const

{

return loggedInUser != nullptr;

}

// Get the current logged-in user

User \*Authentication::getCurrentUser() const

{

return loggedInUser;

}

// Display all users (admin only)

void Authentication::displayAllUsers() const

{

if (loggedInUser && loggedInUser->isAdmin())

{

for (int i = 0; i < users.length(); ++i)

{

users[i].displayInfo();

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

}

}

else

{

cout << "Access denied. Only admins can view all users." << endl;

}

}

// Helper function to find user by username

User \*Authentication::findUserByUsername(const string &username)

{

for (int i = 0; i < users.length(); ++i)

{

if (users[i].getUsername() == username)

{

return &users[i];

}

}

return nullptr;

}

// Load all the users

void Authentication::loadUsersFromFile()

{

FileHandler fileHandler(filename); // Initialize FileHandler with filename

List<string> lines = fileHandler.readAllLines(); // Read all lines from the file

if (lines.length() == 0)

{

cout << "Could not read from the file: " << filename << ". Starting with an empty user list." << endl;

return;

}

// Parse each line into User object and add to the list

for (int i = 0; i < lines.length(); ++i)

{

stringstream ss(lines[i]);

string username, password, email, role, phoneNumber, address, dateOfRegistration;

bool isActive;

ss >> username >> password >> email >> role >> phoneNumber >> address >> dateOfRegistration >> isActive;

User user(username, password, email, role, phoneNumber, address, dateOfRegistration, isActive);

users.push\_back(user); // Add user to the List<User> `users`

}

}

void Authentication::appendUserToFile(const User &user)

{

FileHandler fileHandler(filename); // Initialize FileHandler with filename

// Prepare user data as a single string line

string userData = user.getUsername() + " " + user.getPassword() + " " + user.getEmail() + " " +

user.getRole() + " " + user.getPhoneNumber() + " " + user.getAddress() + " " +

user.getDateOfRegistration() + " " + (user.getIsActive() ? "1" : "0");

// Append the user data line to the file

fileHandler.appendLine(userData);

}

List<User> Authentication::getAllUsers()

{

return this->users;

}

#endif

**user.cpp**  
#ifndef USER

#define USER

#include <iostream>

#include <string>

#include "list.cpp"

#include "text.cpp"

using namespace std;

class User

{

protected:

string username;

string password;

string email;

string role; // "client" or "admin"

string phoneNumber;

string address;

string dateOfRegistration;

bool isActive;

List<string> orderHistory; // Stores order IDs

public:

// Default constructor

User()

: username(""), password(""), email(""), role("client"),

phoneNumber(""), address(""), dateOfRegistration(""), isActive(true) {}

// Constructor

User(string username, string password, string email, string role,

string phoneNumber = "", string address = "", string dateOfRegistration = "", bool isActive = true);

// Copy constructor

User(const User &other)

: username(other.username),

password(other.password),

email(other.email),

role(other.role),

phoneNumber(other.phoneNumber),

address(other.address),

dateOfRegistration(other.dateOfRegistration),

isActive(other.isActive),

orderHistory(other.orderHistory) // Copy the order history

{

}

// Getters and Setters

string getUsername() const;

void setUsername(const string &newUsername);

string getRole() const;

string getPassword() const;

void setPassword(const string &newPassword);

string getEmail() const;

void setEmail(const string &newEmail);

string getPhoneNumber() const;

void setPhoneNumber(const string &newPhoneNumber);

string getAddress() const;

void setAddress(const string &newAddress);

string getDateOfRegistration() const;

void setDateOfRegistration(const string &date);

bool getIsActive() const;

void setIsActive(bool status);

bool isAdmin() const;

bool isClient() const;

// Order History Management

void addOrderToHistory(const string &orderID);

void displayOrderHistory() const;

// Display user information

virtual void displayInfo() const;

static User \*findUserByUsername(const string &username, const List<User> &users)

{

for (int i = 0; i<users.length(); i++)

{

if (users[i].getUsername() == username)

{

return new User(users[i]); // Return a copy of the user

}

}

return nullptr;

}

// Destructor

virtual ~User();

};

// -------- All the definitions of methods ---------

// Constructor

User::User(string username, string password, string email, string role,

string phoneNumber, string address, string dateOfRegistration, bool isActive)

: username(username), password(password), email(email), role(role),

phoneNumber(phoneNumber), address(address), dateOfRegistration(dateOfRegistration), isActive(isActive) {}

// Getters and Setters

string User::getUsername() const { return username; }

void User::setUsername(const string &newUsername) { username = newUsername; }

string User::getPassword() const { return password; }

void User::setPassword(const string &newPassword) { password = newPassword; }

string User::getRole() const { return role; }

string User::getEmail() const { return email; }

void User::setEmail(const string &newEmail) { email = newEmail; }

string User::getPhoneNumber() const { return phoneNumber; }

void User::setPhoneNumber(const string &newPhoneNumber) { phoneNumber = newPhoneNumber; }

string User::getAddress() const { return address; }

void User::setAddress(const string &newAddress) { address = newAddress; }

string User::getDateOfRegistration() const { return dateOfRegistration; }

void User::setDateOfRegistration(const string &date) { dateOfRegistration = date; }

bool User::getIsActive() const { return isActive; }

void User::setIsActive(bool status) { isActive = status; }

bool User::isAdmin() const { return role == "admin"; }

bool User::isClient() const { return role == "client"; }

// Order History Management

void User::addOrderToHistory(const string &orderID)

{

orderHistory.push\_back(orderID);

}

void User::displayOrderHistory() const

{

cout << "Order History for " << username << ":" << endl;

for (int i = 0; i < orderHistory.length(); i++)

{

cout << "- " << orderHistory[i] << endl;

}

}

void User::displayInfo() const

{

// Display username in green color

SingleColorText t(username, Text::green(), 100);

cout << "Username: ";

t.print();

// Display email

t.setText(email);

cout << "Email: ";

t.print();

// Display role

t.setText(role);

cout << "Role: ";

t.print();

// Display phone number

t.setText(phoneNumber);

cout << "Phone Number: ";

t.print();

// Display address

t.setText(address);

cout << "Address: ";

t.print();

// Display date of registration

t.setText(dateOfRegistration);

cout << "Date of Registration: ";

t.print();

// Display active status with dynamic color based on status

t.setText(isActive ? "yes" : "no");

cout << "Active Status: ";

t.setColor(isActive ? Text::green() : Text::red());

t.print();

}

// Destructor

User::~User() {}

#endif

**admin.cpp**  
#ifndef ADMIN

#define ADMIN

#include <iostream>

#include <string>

#include "user.cpp"

#include "product\_manager.cpp"

using namespace std;

class Admin : public User

{

private:

ProductManager productManager;

// Admin functionalities

void displayCatalog() const;

void addProduct();

void removeProduct();

void updateProduct();

public:

Admin(string username, string password, string email, string phoneNumber = "", string address = "", string dateOfRegistration = "");

Admin();

// Initiate the admin panel

void begin();

~Admin();

};

Admin::Admin() : productManager("files/product.txt"){}

Admin::Admin(string username, string password, string email, string phoneNumber, string address, string dateOfRegistration)

: User(username, password, email, "admin", phoneNumber, address, dateOfRegistration),

productManager("files/product.txt"){}

void Admin::displayCatalog() const

{

productManager.displayCatalog();

}

void Admin::addProduct()

{

string productID, productName, description;

double price;

int quantity;

// Setting heading

SingleColorText s("\nAdd a new product", Text::yellow());

s.setBold(1);

s.print();

cout << endl;

// Taking the product id

s.setText("Enter product ID: ");

s.setBold(0);

s.setColor(Text::blue());

s.print();

cin >> productID;

s.setText("Enter product name: ");

s.print();

cin.ignore(); // To handle newline left in the input buffer

getline(cin, productName);

s.setText("Enter product price: ");

s.print();

cin >> price;

s.setText("Enter quantity: ");

s.print();

cin >> quantity;

s.setText("Enter description: ");

s.print();

cin.ignore();

getline(cin, description);

Product newProduct(productID, productName, price, quantity, description);

productManager.addProduct(newProduct);

}

void Admin::removeProduct()

{

SingleColorText s("Enter product ID to remove: ", Text::blue());

s.print();

string productID;

cin >> productID;

if (productManager.removeProduct(productID))

{

s.setText("Product removed successfully.\n");

s.setColor(Text::green());

s.print();

}

else

{

s.setText("Product not found.\n");

s.setColor(Text::red());

s.print();

}

}

void Admin::updateProduct()

{

string productID, productName, description;

double price;

int quantity;

// Setting heading

SingleColorText s("\nAdd a new product", Text::yellow());

s.setBold(1);

s.print();

cout << endl;

// Taking the product id

s.setText("Enter product ID: ");

s.setBold(0);

s.setColor(Text::blue());

s.print();

cin >> productID;

s.setText("Enter product name: ");

s.print();

cin.ignore(); // To handle newline left in the input buffer

getline(cin, productName);

s.setText("Enter product price: ");

s.print();

cin >> price;

s.setText("Enter quantity: ");

s.print();

cin >> quantity;

s.setText("Enter description: ");

s.print();

cin.ignore();

getline(cin, description);

Product updatedProduct(productID, productName, price, quantity, description);

if (productManager.updateProduct(productID, updatedProduct))

{

s.setText("Product updated successfully.\n");

s.setColor(Text::green());

s.print();

}

else

{

s.setText("Product not found.\n");

s.setColor(Text::red());

s.print();

}

}

void Admin::begin()

{

int choice;

do

{

List<string> colors(3);

colors.push\_back(Text::red());

colors.push\_back(Text::green());

colors.push\_back(Text::blue());

MultiColorText m("Admin Panel", colors);

cout << "\n---------- ";

m.print();

cout << " ----------\n";

cout << "1. Display Catalog\n";

cout << "2. Add Product\n";

cout << "3. Remove Product\n";

cout << "4. Update Product\n";

cout << "0. Exit Admin Panel\n";

SingleColorText s("", Text::blue());

s.setText("Enter your choice: ");

s.setDelay(30);

s.print();

cin >> choice;

switch (choice)

{

case 1:

displayCatalog();

break;

case 2:

addProduct();

break;

case 3:

removeProduct();

break;

case 4:

updateProduct();

break;

case 0:

cout << "Exiting Admin Panel...\n";

break;

default:

s.setColor(Text::red());

s.setText("Invalid choice. Please try again.");

cout << endl;

s.print();

cout << endl;

}

} while (choice != 0);

}

Admin::~Admin() {}

#endif

**client.cpp**  
#ifndef CLIENT

#define CLIENT

#include <iostream>

#include "user.cpp"

#include "cart.cpp"

#include "product.cpp"

#include "user.cpp"

using namespace std;

class Client

{

private:

Cart cart;

User \*user;

public:

// Default constructor

Client() {}

Client(User \*user)

{

this->user = user;

cart.setUsername(user->getUsername());

}

// Method to display the client panel

void begin()

{

SingleColorText s("", Text::blue());

int choice;

do

{

cout << "\nWelcome, " << Text::green() << user->getUsername() << Text::white() << "! Choose an option:" << endl;

cout << "1. View Products" << endl;

cout << "2. Go to Cart" << endl;

cout << "3. Add Product to Cart" << endl;

cout << "4. Update Profile" << endl;

cout << "0. Exit" << endl;

s.setText("Enter your choice: ");

s.print();

cin >> choice;

switch (choice)

{

case 1:

viewProducts();

break;

case 2:

goToCart();

break;

case 3:

{

ProductManager pm("files/product.txt");

List<Product> products = pm.getAllProducts();

string productID;

int quantity;

cout << "Enter the Product ID to add to cart (or type 'exit' to return): ";

cin >> productID;

if (productID == "exit")

return;

cout << "Enter the quantity: ";

cin >> quantity;

for (int i = 0; i < products.length(); i++)

{

if (products[i].getProductID() == productID)

{

cart.addProduct(products[i]);

}

}

}

break;

case 4:

updateProfile();

break;

case 0:

s.setText("Exiting the client panel. Goodbye, " + user->getUsername() + "!");

s.setColor(Text::green());

s.print();

break;

default:

s.setText("Invalid choice. Please try again.");

s.setColor(Text::red());

s.print();

break;

}

} while (choice != 0);

}

private:

// Method to view available products

void viewProducts()

{

ProductManager pm("files/product.txt");

pm.displayCatalog();

cout << endl;

List<Product> products = pm.getAllProducts();

string productID;

cout << "Enter the Product ID to add to cart (or type 'exit' to return): ";

cin >> productID;

if (productID == "exit")

return;

for (int i = 0; i < products.length(); i++)

{

if (products[i].getProductID() == productID)

{

cart.addProduct(products[i]);

return;

}

}

cout << "Product ID not found. Returning to menu." << endl;

}

// Method to manage the cart

void goToCart()

{

SingleColorText s("", Text::blue());

int choice;

do

{

cout << "\nCart Menu:" << endl;

cout << "1. View Cart" << endl;

cout << "2. Remove Item from Cart" << endl;

cout << "3. Checkout" << endl;

cout << "4. Return to Main Menu" << endl;

s.setText("Enter your choice: ");

s.print();

cin >> choice;

switch (choice)

{

case 1:

cart.displayCart();

break;

case 2:

{

string productID;

s.setText("Enter the Product ID to remove: ");

s.setColor(Text::blue());

s.print();

cin >> productID;

cart.removeProduct(productID);

break;

}

case 3:

checkout();

break;

case 4:

cout << "Returning to main menu..." << endl;

break;

default:

s.setColor(Text::red());

s.setText("Invalid choice, Please try again.");

cout << endl;

s.print();

cout << endl;

break;

}

} while (choice != 4);

}

// Method to checkout

void checkout()

{

double totalCost = cart.getTotalPrice();

cout << "\nTotal Cost: $" << totalCost << endl;

string confirmation;

SingleColorText s("Do you want to confirm the purchase? (yes/no): ", Text::blue());

s.print();

cin >> confirmation;

int amount;

if (confirmation == "yes")

{

s.setText("Enter the amount: ");

s.setColor(Text::blue());

s.print();

cin >> amount;

if (amount == totalCost)

{

s.setText("Thanks for purchasing.");

s.setColor(Text::green());

s.print();

}

else if (amount > totalCost)

{

s.setText("Thanks for purchasing, " + to\_string(amount - totalCost) + " added back to your account.");

s.setColor(Text::green());

s.print();

}

else

{

s.setText("Thanks for purchasing, " + to\_string(totalCost - amount) + " will be taken as cash on delivery.");

s.setColor(Text::green());

s.print();

}

cart.items.clear();

}

else{

s.setText("Checkout canceled.");

s.setColor(Text::red());

s.print();

}

}

// Method to update client profile

void updateProfile()

{

cout << "\nUpdate Profile:" << endl;

cout << "1. Update Phone Number" << endl;

cout << "2. Update Address" << endl;

cout << "3. Return to Main Menu" << endl;

string username, address;

int choice;

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter new phone number: ";

cin >> username;

user->setUsername(username);

cout << "Phone number updated!" << endl;

break;

case 2:

cout << "Enter new address: ";

cin.ignore();

getline(cin, address);

user->setAddress(address);

cout << "Address updated!" << endl;

break;

case 3:

cout << "Returning to main menu." << endl;

break;

default:

cout << "Invalid choice. Returning to menu." << endl;

break;

}

}

};

#endif

**product\_manager.cpp**

#ifndef PRODUCT\_MANAGER

#define PRODUCT\_MANAGER

#include "list.cpp"

#include "file handler.cpp"

#include "product.cpp"

#include <iostream>

using namespace std;

class ProductManager

{

private:

List<Product> productCatalog;

FileHandler fileHandler;

public:

ProductManager(const string &filePath);

void loadProductsFromFile();

void saveProductsToFile();

void addProduct(const Product &product);

bool removeProduct(const string &productID);

void displayCatalog() const;

bool updateProduct(const string &productID, const Product &updatedProduct);

List<Product> getAllProducts();

};

ProductManager::ProductManager(const string &filePath) : fileHandler(filePath)

{

loadProductsFromFile();

}

void ProductManager::loadProductsFromFile()

{

List<string> lines = fileHandler.readAllLines();

for (int i = 0; i < lines.length(); ++i)

{

string line = lines.get(i);

size\_t pos = 0;

string productID, productName, description;

double price;

int quantity;

pos = line.find(",");

productID = line.substr(0, pos);

line.erase(0, pos + 1);

pos = line.find(",");

productName = line.substr(0, pos);

line.erase(0, pos + 1);

pos = line.find(",");

price = stod(line.substr(0, pos));

line.erase(0, pos + 1);

pos = line.find(",");

quantity = stoi(line.substr(0, pos));

line.erase(0, pos + 1);

description = line;

Product product(productID, productName, price, quantity, description);

productCatalog.push\_back(product);

}

}

void ProductManager::saveProductsToFile()

{

List<string> lines;

for (int i = 0; i < productCatalog.length(); ++i)

{

Product &product = productCatalog[i];

string line = product.getProductID() + "," + product.getProductName() + "," +

to\_string(product.getPrice()) + "," + to\_string(product.getQuantity()) + "," +

product.getDescription();

lines.push\_back(line);

}

fileHandler.writeAllLines(lines);

}

void ProductManager::addProduct(const Product &product)

{

productCatalog.push\_back(product);

saveProductsToFile();

}

bool ProductManager::removeProduct(const string &productID)

{

for (int i = 0; i < productCatalog.length(); ++i)

{

if (productCatalog[i].getProductID() == productID)

{

productCatalog.removeAt(i);

saveProductsToFile();

return true;

}

}

return false;

}

void ProductManager::displayCatalog() const

{

if (productCatalog.length() <= 0)

{

cout << "No products to display.\n";

}

else

{

for (int i = 0; i < productCatalog.length(); ++i)

{

productCatalog[i].display();

cout << endl;

}

}

}

bool ProductManager::updateProduct(const string &productID, const Product &updatedProduct)

{

for (int i = 0; i < productCatalog.length(); ++i)

{

if (productCatalog[i].getProductID() == productID)

{

productCatalog[i] = updatedProduct;

saveProductsToFile();

return true;

}

}

return false;

}

List<Product> ProductManager::getAllProducts(){

return productCatalog;

}

#endif

**product.cpp**

#ifndef PRODUCT

#define PRODUCT

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Product

{

private:

string productID; // Unique identifier for the product

string productName; // Name of the product

double price; // Price of the product

int quantity; // Quantity available in stock

string description; // Description of the product

public:

// Default constructor

Product() : productID(""), productName(""), price(0.0), quantity(0), description("") {}

// Parameterized constructor

Product(string productID, string productName, double price, int quantity, string description)

: productID(productID), productName(productName), price(price), quantity(quantity), description(description) {}

// Getter and Setter methods

string getProductID() const { return productID; }

void setProductID(const string &id) { productID = id; }

string getProductName() const { return productName; }

void setProductName(const string &name) { productName = name; }

double getPrice() const { return price; }

void setPrice(double p) { price = p; }

int getQuantity() const { return quantity; }

void setQuantity(int qty) { quantity = qty; }

string getDescription() const { return description; }

void setDescription(const string &desc) { description = desc; }

// Method to display product details

void displayProductDetails() const

{

cout << "Product ID: " << productID << endl;

cout << "Product Name: " << productName << endl;

cout << "Price: $" << price << endl;

cout << "Quantity Available: " << quantity << endl;

cout << "Description: " << description << endl;

}

// Method to check if product is in stock

bool isInStock() const

{

return quantity > 0;

}

// Method to update stock (after purchase)

void updateStock(int purchasedQuantity)

{

if (quantity >= purchasedQuantity)

{

quantity -= purchasedQuantity;

cout << "Stock updated. Remaining stock: " << quantity << endl;

}

else

{

cout << "Not enough stock available!" << endl;

}

}

// Method to restock the product

void restock(int additionalQuantity)

{

quantity += additionalQuantity;

cout << "Product restocked. New stock: " << quantity << endl;

}

void display()

{

int width = max(productName.length(), description.length()) + 17;

string line;

for (int i = 0; i < width - 2; i++)

{

line += '\_';

}

// creating object of single color text and initializing with line

SingleColorText s(line, Text::cyan(), 1, 10);

// printing horizontal line

cout << Text::cyan() << Text::bold() << '/';

s.print();

cout << Text::bold() << Text::cyan() <<"\\" << endl;

// printing empty line

cout << '\\';

emptyLine(width);

cout << '/' << endl;

// Printing product id

cout << "/ " << Text::white();

s.setText(productID);

s.setDelay(100);

s.print();

printSpaces(width - productID.length() - 3);

cout << Text::cyan() << Text::bold() << "\\" << endl;

// printing horizontal line

cout << Text::cyan() << Text::bold() << '\\';

s.setText(line);

s.setDelay(10);

s.print();

cout << Text::bold() << Text::cyan() <<"/" << endl;

// printing empty line

cout << '/';

emptyLine(width);

cout << '\\' << endl;

// Printing name

cout << "\\ " << Text::white() << "Name";

printSpaces(7);

cout << ": " << Text::normal();

s.setBold(0);

s.setText(productName);

s.setDelay(50);

s.print();

printSpaces(width - 17 - productName.length());

cout << Text::cyan() << Text::bold() << " /" << endl;

// Printing Price

cout << "/ " << Text::white() << "Price";

printSpaces(6);

cout << ": " << Text::normal() << Text::cyan() << fixed << setprecision(0) << price;

printSpaces(width - 17 - numberOfDigits(price));

cout << Text::cyan() << Text::bold() << " \\" << endl;

// printing quantity

cout << "\\ " << Text::white() << "Quantity";

printSpaces(3);

cout << ": " << Text::normal() << Text::cyan() << quantity;

printSpaces(width - 17 - numberOfDigits(quantity));

cout << Text::cyan() << Text::bold() << " /" << endl;

// Printing

cout << Text::cyan() << "/ " << "\033[37m" << "Description: " << Text::normal();

s.setText(description);

s.print();

cout << Text::cyan() << Text::bold() << " \\" << endl;

// printing horizontal line

cout << Text::cyan() << Text::bold() << '\\';

s.setText(line);

s.setDelay(1);

s.setBold(1);

s.print();

cout << Text::bold() << Text::cyan() <<"/" << endl << endl << Text::white() << Text::normal();

}

private:

void emptyLine(int width)

{

cout << Text::bold();

for (int i = 0; i < width - 2; i++)

{

cout << ' ';

}

}

void printSpaces(int times)

{

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

{

cout << ' ';

}

}

int numberOfDigits(int n)

{

int cnt = 0;

while (n > 0)

{

cnt++;

n /= 10;

}

return cnt;

}

public:

// Destructor

~Product() {}

};

#endif

**cart.cpp**

#ifndef CART

#define CART

#include <iostream>

#include <fstream>

#include <string>

#include "product.cpp"

#include "list.cpp"

#include "file handler.cpp"

using namespace std;

class Cart

{

private:

List<Product> items; // List of products in the cart

string userID; // User-specific file ID for persistence

void saveCart() const;

void loadCart();

public:

Cart() {}

// Constructor that initializes cart for a user and loads existing items

Cart(const string &userID) : userID(userID)

{

loadCart();

}

// Add a product to the cart

void addProduct(const Product &product);

// Remove a product from the cart by product ID

void removeProduct(const string &productID);

// Display items in the cart

void displayCart() const;

// Calculate the total price of items in the cart

double getTotalPrice() const;

// Clear the cart

void clearCart();

void setUsername(const string &username);

};

// Helper method to load cart items from the user's file

void Cart::loadCart()

{

// Initialize FileHandler with the user's cart file path

FileHandler fileHandler("files/cart/" + userID + ".txt");

// Read all lines from the user's cart file

List<string> lines = fileHandler.readAllLines();

// If no lines were read, start with an empty cart

if (lines.length() == 0)

{

cout << "Could not open file for user: " << userID << ". Starting with an empty cart." << endl;

return;

}

// Parse each line to extract product details and add to items list

for (int i = 0; i < lines.length(); ++i)

{

istringstream lineStream(lines[i]);

string id, name, description;

double price;

int quantity;

// Read product details from the line

lineStream >> id >> name >> price >> quantity;

getline(lineStream, description); // Read the remaining description

// Create Product object and add it to the cart items

Product product(id, name, price, quantity, description);

items.push\_back(product);

}

}

// Helper method to save cart items to the user's file

void Cart::saveCart() const

{

// Create a FileHandler instance for the user's cart file

FileHandler fileHandler("files/cart/" + userID + ".txt");

// Prepare a List<string> to hold the formatted lines for each product

List<string> lines;

// Format each product's details and add to lines

for (int i = 0; i < items.length(); ++i)

{

Product p = items.get(i);

string line = p.getProductID() + " " +

p.getProductName() + " " +

to\_string(p.getPrice()) + " " +

to\_string(p.getQuantity()) + " " +

p.getDescription();

lines.push\_back(line);

}

// Write all lines to the file using FileHandler

fileHandler.writeAllLines(lines);

}

// Add a product to the cart

void Cart::addProduct(const Product &product)

{

items.push\_back(product);

cout << product.getProductName() << " added to cart." << endl;

saveCart(); // Save after each change

}

// Remove a product from the cart by product ID

void Cart::removeProduct(const string &productID)

{

for (int i = 0; i < items.length(); ++i)

{

if (items.get(i).getProductID() == productID)

{

cout << items.get(i).getProductName() << " removed from cart." << endl;

items.removeAt(i);

saveCart(); // Save after each change

return;

}

}

cout << "Product with ID " << productID << " not found in cart." << endl;

}

// Display items in the cart

void Cart::displayCart() const

{

if (items.length() == 0)

{

cout << "Your cart is empty." << endl;

return;

}

cout << "Items in your cart:" << endl;

for (int i = 0; i < items.length(); ++i)

{

items.get(i).display();

}

}

// Calculate the total price of items in the cart

double Cart::getTotalPrice() const

{

double total = 0;

for (int i = 0; i < items.length(); ++i)

{

total += items.get(i).getPrice();

}

return total;

}

// Clear the cart

void Cart::clearCart()

{

items.clear();

cout << "Your cart has been cleared." << endl;

saveCart(); // Save the empty state

}

void Cart::setUsername(const string &username)

{

this->userID = username;

}

#endif

**file handler.cpp**

#ifndef FILE\_HANDLER

#define FILE\_HANDLER

#include <iostream>

#include <fstream>

#include <string>

#include "list.cpp"

using namespace std;

class FileHandler {

private:

string filename; // File path for storing data

public:

// Constructor to initialize filename

FileHandler(const string &filename) : filename(filename) {}

// Read all lines from the file into a List

List<string> readAllLines() {

List<string> lines;

ifstream file(filename);

if (!file.is\_open()) {

cout << "Error: Could not open file " << filename << endl;

return lines;

}

string line;

while (getline(file, line)) {

lines.push\_back(line); // Assuming List has push\_back method

}

file.close();

return lines;

}

// Write all lines from a List to the file (overwrites file contents)

void writeAllLines(const List<string> &lines) {

ofstream file(filename, ios::trunc);

if (!file.is\_open()) {

cout << "Error: Could not open file " << filename << endl;

return;

}

for (int i = 0; i < lines.length(); ++i) {

file << lines[i] << endl;

}

file.close();

}

// Append a single line to the file

void appendLine(const string &line) {

ofstream file(filename, ios::app);

if (!file.is\_open()) {

cout << "Error: Could not open file " << filename << endl;

return;

}

file << line << endl;

file.close();

}

// Append multiple lines from a List to the file

void appendLines(const List<string> &lines) {

ofstream file(filename, ios::app);

if (!file.is\_open()) {

cout << "Error: Could not open file " << filename << endl;

return;

}

for (int i = 0; i < lines.length(); ++i) {

file << lines[i] << endl;

}

file.close();

}

};

#endif

**list.cpp**

#ifndef LIST

#define LIST

#include <iostream>

#include <stdexcept> // For out\_of\_range exception

using namespace std;

template <typename T>

class List

{

T \*arr;

int index;

int size;

public:

List(int size = 10); // Constructor

List(const List<T> &other); // Copy Constructor

bool push\_back(T elem); // Push element to the end

bool push\_front(T elem); // Push element to the start

void pop\_last(); // Remove the last element

bool pop\_start(); // Remove the first element

bool removeAt(int i);

T get(int i) const; // Get the element at index i

int length() const; // Get the current length of the list

void print(); // Print the list

void clear();

T &operator[](int i) const; // Access element by index

~List(); // Destructor

private:

void moveValuesToRight(); // Move values to the right (for push\_front)

void moveValuesToLeft(); // Move values to the left (for pop\_start)

};

// Constructor definition

template <typename T>

List<T>::List(int size) : size(size), index(-1)

{

arr = new T[size];

}

// Copy constructor definition

template <typename T>

List<T>::List(const List<T> &other) : size(other.size), index(other.index)

{

arr = new T[size];

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

{

arr[i] = other.arr[i]; // Deep copy elements

}

}

// push\_back definition

template <typename T>

bool List<T>::push\_back(T elem)

{

if (index == size - 1)

{

cout << "List is full" << endl;

return false;

}

arr[++index] = elem;

return true;

}

// push\_front definition

template <typename T>

bool List<T>::push\_front(T elem)

{

if (index == size - 1)

{

cout << "List is full" << endl;

return false;

}

moveValuesToRight();

arr[0] = elem;

index++;

return true;

}

// pop\_last definition

template <typename T>

void List<T>::pop\_last()

{

if (index == -1)

{

cout << "List is empty." << endl;

return;

}

index--; // Decrease index to remove the last element

}

// pop\_start definition

template <typename T>

bool List<T>::pop\_start()

{

if (index == -1)

{

cout << "List is empty." << endl;

return false;

}

moveValuesToLeft();

index--;

return true;

}

// removeAt definition

template <typename T>

bool List<T>::removeAt(int i)

{

if (i < 0 || i > index)

{

cout << "Index out of bounds" << endl;

return false;

}

// Shift elements to the left, starting from the index to remove

for (int j = i; j < index; j++)

{

arr[j] = arr[j + 1];

}

index--; // Decrease the index to reflect the removed element

return true;

}

// get definition

template <typename T>

T List<T>::get(int i) const

{

if (i < 0 || i > index)

{

throw std::out\_of\_range("Index out of bounds");

}

return arr[i];

}

// length definition

template <typename T>

int List<T>::length() const

{

return index + 1;

}

// print definition

template <typename T>

void List<T>::print()

{

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

{

cout << arr[i] << ' ';

}

cout << endl;

}

// clear definition

template <typename T>

void List<T>::clear()

{

index = -1; // Reset index to -1, effectively clearing the list

}

// operator[] definition

template <typename T>

T &List<T>::operator[](int i) const

{

if (i < 0 || i > index)

{

throw std::out\_of\_range("Index out of bounds");

}

return arr[i];

}

// Destructor definition

template <typename T>

List<T>::~List()

{

delete[] arr;

}

// moveValuesToRight definition

template <typename T>

void List<T>::moveValuesToRight()

{

for (int i = index; i >= 0; i--)

{

arr[i + 1] = arr[i];

}

}

// moveValuesToLeft definition

template <typename T>

void List<T>::moveValuesToLeft()

{

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

{

arr[i] = arr[i + 1];

}

}

#endif

**text.cpp**

#ifndef TEXT

#define TEXT

#include <iostream>

#include <string>

#include <thread>

#include <chrono>

#include <cstdlib> // For rand()

#include <ctime> // For time()

#include "list.cpp"

using namespace std;

// Base class for all text types

class Text

{

protected:

string text;

int delay;

public:

// Constructor to initialize text and delay

Text(const string &text, int delay = 100) : text(text), delay(delay) {}

// Virtual print method (to be overridden by derived classes)

virtual void print() = 0;

// Static method for reset

static string reset() { return "\033[0m"; }

// Static methods for ANSI color codes

static string red() { return "\033[31m"; }

static string green() { return "\033[32m"; }

static string blue() { return "\033[34m"; }

static string yellow() { return "\033[33m"; }

static string magenta() { return "\033[35m"; }

static string cyan() { return "\033[36m"; }

static string white() { return "\033[37m"; }

static string bold() { return "\033[1m"; }

static string normal() { return "\033[0m"; }

};

// Derived class for single-color text

class SingleColorText : public Text

{

private:

string color; // Single color for the text

bool bold;

public:

// Constructor for single-color text

SingleColorText(const string &text, const string &color, bool bold = 0, int delay = 40)

: Text(text, delay), color(color), bold(bold) {}

void setText(const string text)

{

this->text = text;

}

void setBold(const bool bold)

{

this->bold = bold;

}

void setDelay(const int delay)

{

this->delay = delay;

}

void setColor(const string &color)

{

this->color = color;

}

// Method to display the animated text in single color

void print() override

{

for (size\_t i = 0; i < text.length(); ++i)

{

if (bold)

cout << Text::bold();

else

cout << Text ::normal();

cout << color << text[i] << Text::reset(); // Call reset from Text

cout.flush();

this\_thread::sleep\_for(chrono::milliseconds(delay));

}

}

};

// Derived class for multi-color text

class MultiColorText : public Text

{

private:

List<string> colors; // List to store multiple colors

public:

// Constructor for multi-color text

MultiColorText(const string &text, const List<string> &colors, int delay = 100)

: Text(text, delay), colors(colors)

{

}

void setText(const string text)

{

this->text = text;

}

void setColor(const List<string> &colors)

{

this->colors = colors;

}

// Method to display the animated text with multiple colors

void print() override

{

int colorCount = colors.length();

for (int i = 0; i < text.length(); ++i)

{

string color = colors[i % colorCount];

cout << color << text[i] << Text::reset(); // Call reset from Text

cout.flush();

this\_thread::sleep\_for(chrono::milliseconds(delay));

}

}

};

class SpiralText : public Text

{

public:

SpiralText(const string &text, int delay = 100) : Text(text, delay) {}

void print() override

{

int n = text.length();

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

{

// Move the cursor to a new position to create the spiral effect

cout << string(i, ' ') << text[i] << flush;

this\_thread::sleep\_for(chrono::milliseconds(delay));

cout << endl;

}

}

};

class RandomColorText : public Text

{

public:

RandomColorText(const string &text, int delay = 100) : Text(text, delay)

{

srand(time(0)); // Seed the random number generator

}

string randomColor()

{

int color = rand() % 5; // Random number between 0 and 4

switch (color)

{

case 0:

return Text::red();

case 1:

return Text::green();

case 2:

return Text::blue();

case 3:

return Text::yellow();

case 4:

return Text::cyan();

default:

return Text::reset();

}

}

void print() override

{

for (size\_t i = 0; i < text.length(); ++i)

{

cout << randomColor() << text[i] << Text::reset();

cout.flush();

this\_thread::sleep\_for(chrono::milliseconds(delay));

}

cout << endl;

}

};

class BlinkingText : public Text

{

int times;

public:

BlinkingText(const string &text, int delay = 100, int times = 10) : Text(text, delay), times(times) {}

void print() override

{

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

{

// Print the text

cout << text << flush;

this\_thread::sleep\_for(chrono::milliseconds(delay));

// Clear the text

cout << "\r" << string(text.length(), ' ') << "\r" << flush;

this\_thread::sleep\_for(chrono::milliseconds(delay));

}

}

};

#endif

**| - - - - - End - - - - - |**