Bike Rental System

Title Page

- **Course Name:** Basics of Programming 2
- **Course ID:** [**BMEVIIIAA03**]
- **Full Name:** [Mahmoud Marzouk Elkadeem]
- **Neptun Code:** [MR8BRM]
- **Lab Group ID:** [Your Lab Group ID]
- **Lab Instructor:** Dr. Vaitkus Márton
- **Project Title:** Bike Rental Management System
- **Project Type:** Final Project

Introduction

This project involves the design and implementation of a Bike rental system in C++. The system allows customers to rent and return bikes while the admins will manage the bikes inventory and the information of the rentals and the bikes . and the goal is to simulate a real rental service that keeps tracking the bike and the duration of the renting and the fees.

Problem Definition :

In a real rental business of bikes , it is necessary to manage the inventory of the bikes and keep tracking the rentals and the costs and the time even after closing the program and linking between the customers and their rented bikes.

Suggested solutions:

This Project provides a system where admin can add and delete and search for any bike and viewing the rentals , while customers can rent and return any bike and the system keeps tracking the time and calculates the costs and it stores the data for the bike and the customers using file handling .

Program interface

Starting the Program:

1: open your C++ IDE

2: compile the main.cpp file and the Bikes.h and the Bikes.cpp files

For terminating the Program:

For the admin and the customer, their menus includes an option to log out.

From the main menu, select the Exit option and confirm your Exit and it will terminate the program if yes. All data related to the bikes and the customers are automatically saved to the file bikes_data.txt and user_data.txt .

Program Execution

Once we run the program, the main menu will be displayed and showing the following

Main Menu Options:

- **Login:** used if the customer wants to login and the same for admin
- **Customer registration:** if a new customer wants to be a part of the system
- **Exit Program:** For Exiting from the system and there is confirmation before that

Admin Menu:

- **displaying all the bikes:** this will display all the data of the bikes
- **Searching for a bike**: searching for any bike by ID or brand and then it will show all the details of this bike
- **Adding a new bike:** for creating a new bike to the system and the admin will be asked to enter the data of this bike like the brand and the type and so on
- **Deleting a bike:** for deleting a bike from the system in which the user will be asked to enter the ID of this bike and it will be deleted completely from the system if it is found
- **Viewing the active rentals:** for displaying the active rentals and their rented bikes

- **logging out:** for logging out from the menu and back to main menu

Customer Menu:

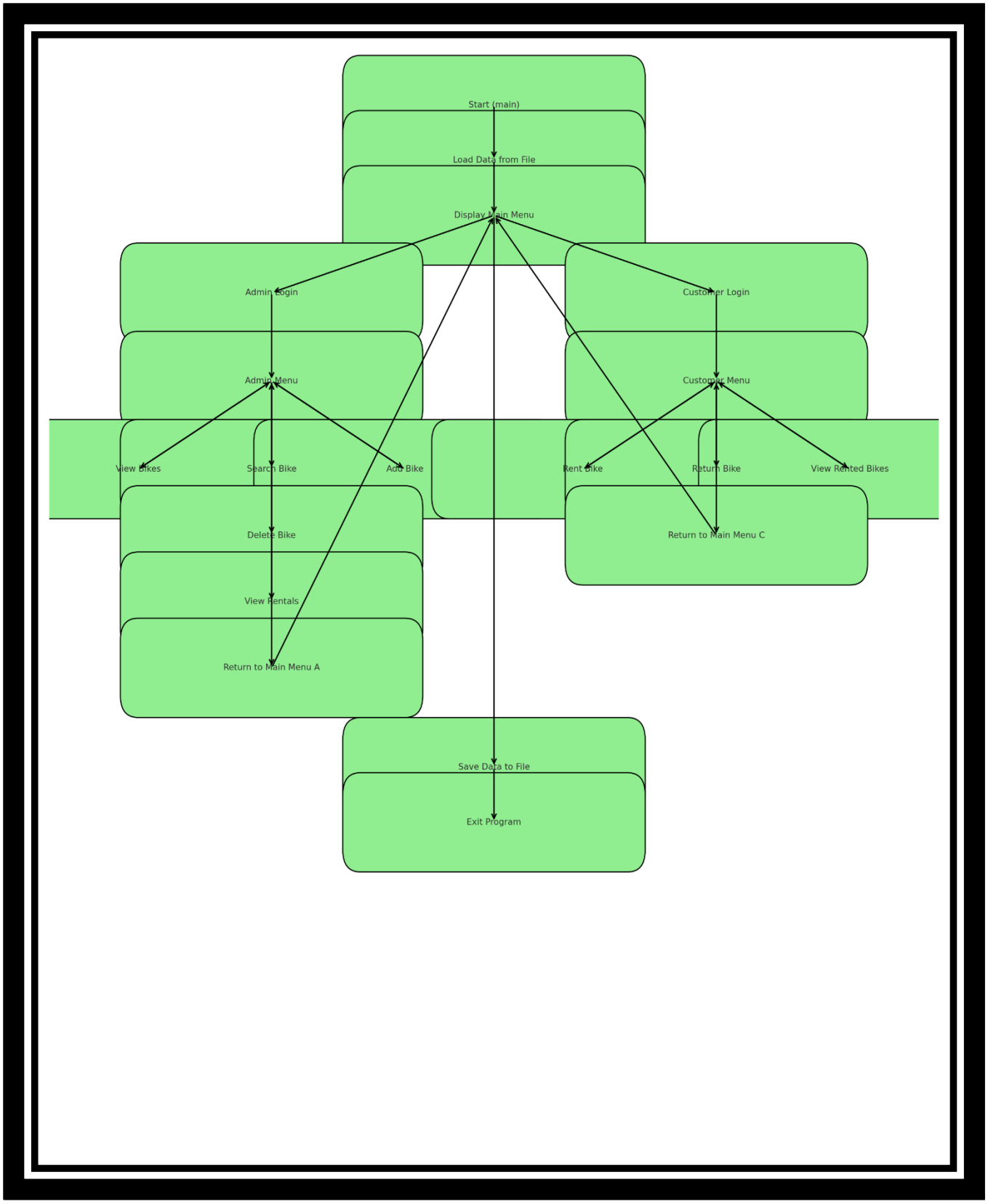
- **Renting a bike:** for renting a bike if it is available and once it is rented, the timer will keep counting even if he exits from the program because we are storing the data of the time for each time he runs the program, and the bike is being rented.

- **Returning a bike:** for returning the bike and calculating the rental fees

- **Viewing the Rented bikes:** for displaying all the active bikes of the customer

- **logging out:** for logging out from the menu and back to main menu

visual image for showing the project is working



inputs and outputs

inputs:

- Integer values for selecting from any menu options.

- Strings for entering the data of the customer (username, password), integer value for the user ID**,** and the same inputs are used for logging to access the menu of the admin and the customer.

- Strings for entering the data of each bike (type, brand, model and size) and bikeID (integer) which are used for the selection of the bike to delete,search,rent, return.

outputs:

- All the bikes which are on the system with their details and their availability status

- Details of bikes after searching for it.

- Rental confirmation with start time.

- Return confirmation with the rental duration and cost.

- Viewing the active rentals and their users IDS

Some Examples for the inputs and the outputs

Example Input:

Enter Bike Brand: Giant ::: Enter Bike ID: 101 ::: Enter your choice : 2

Enter the password: mer23

Example Output:

Bike successfully added! ::: Bike is rented successfully! Time is running

Bike … is deleted successfully.

File I/O:

- **Data is stored in bikes_data.txt with the format:**

brand,model,ID,type,size,price,mileage,availability,(the data of the time for tracking the renting)

Program Structure

overall architecture:

The program is structured using three main classes Bike, bikesystem, user, usermanagement

Bike class: will represent the bikes with their attributes and some necessary methods for each bike

bikesystem class: will manage all the functions of the system like adding , returning , saving to files, loading form files (all the menu options of both the customer and the admin menus)

user class will represent the attributes of users like the name, password and ID and it will Also manage some functions for verification.

Class: `Bike`

`Attributes:`

- `brand`, `model`, `type`, `size` → strings
- `id` → int
- `isAvailable` → bool
- `price`, `mileage`, `offlinePeriod`, `previousTime` → doubles
- `lastShutdownTime`, `programRestartTime` → time_t

`Responsibilities:`

- `Store and display bike details.`
- `Track availability.`
- `Track and update mileage.`
- `Record shutdown and restart times.`
- `Calculate offline duration.`
- `Track previous runtime session duration.`

Methods:

Getters and Setters:

- **Setters:** for
  brand, model, type, size, price, mileage, availability, offlineperiod, previoustime
- **Getters:** for
  brand, model, bikeID, type, size, availability, price, mileage, previoustime, offlineperiod, lastshutdowntime

Constructor

- bike(string br, string mod, int NUM, string ty, string size, double cost)

Time Tracking Functions

- recordshutdowntime()
- getlastshutdowntime() const
- recordrestarttime()
- loading_time(time_t shutdown_t)
- startrenting()
- getduration()
- setprevioustime(double seconds)
- getprevioustime()

Rental Status Control

- setrented() → mark bike as rented
- setreturning() → mark bike as returned
- resetrentedstate() → reset renting state

Class bikesystem:

Data Members:

- vector<bike*> bikes → dynamic array of bike pointers
- map<int, int> customerBIKE → map of customer IDs to bike IDs (tracking rentals)

system functions

addbike() → adds a new bike to the system

createbike() → constructs and returns a new bike object

deletingbike(const int ID) → removes a bike from the system by ID

savingbike() const → saves current bike data to file

loadingbikes() → loads bikes from file into system

confirmexit() → checks for exit confirmation

~bikesystem() → destructor to clean up dynamic memory

For displaying and searching

displaybikes() const → displays all bikes

printbikedeatails(const bike* a) const → prints details of a specific bike

findbikebyID(const int ID) const → retrieves a bike by its ID

showavaiablebikes() const → displays available bikes

searchingbyID(const int ID) const → searches for a bike by ID

searchingbyBRAND(const string& b) const → searches bikes by brand

Rental Operations

- rentingbike(const int ID, const int userID = 0) → rent a bike to a user
- returningbike(const int ID, const int userID = 0) → return a rented bike
- veiwingRentals() → view current bike rentals

Helper / Validation

- bikeIDexist(int number) → check if a bike ID already exists

Global Function

- lookingfor_Bike(const bikesystem& search) → utility function for bike searching

Base Class: user

Attributes:

- string name → user's name
- string password → user's password
- int userID → unique identifier

Core Functions:

- **Constructor:**
  user(string n, string p, int ID) → initialize user details
- **Getters:**
  - string getname()
  - string getpassword()
  - int getuserID()
- **Validation:**
  - bool checkname(string username)
  - bool checkpassword(string pass) const
- **Virtual / Abstract Functions (must be overridden):**
  - virtual void showmenu(bikesystem& system) = 0
  - virtual bool isadmin()
  - virtual vector<int> getrentedbikesID() const = 0
- **Destructor:**
  virtual ~user()

Derived Class: admin (inherits from user)

Functions:

- **Constructor:**
  admin(string n, string pw, int ID)
- **Overrides:**

void showmenu(bikesystem& obj) override , this will show the menu of the admin that include adding and displaying etc …

bool isadmin()

vector<int> getrentedbikesID() const override
(has to be implemented to avoid making admin an abstract class)

Derived Class: customer (inherits from user)

Attributes:

- vector<int> rentedbikesID → stores IDs of bikes currently rented by this customer

Functions:

- **Constructor:**
  customer(string n, string ps, int ID)
- **Rental Operations:**
  - void addrental(int bikeId) → add a bike to customer's rentals
  - void removeRental(int bikeID) → remove a bike from customer's rentals
- **Overrides:**
  - void showmenu(bikesystem& obj) override
  - vector<int> getrentedbikesID() const override

class usermanagement:

Attributes:

- vector<user*> users
  → dynamic vector storing pointers of the user class objects to access both customer and admin objects.

Methods:

User Registration:

- customer* registercustomer(string name, string password, int ID)
  → creates and registers a new customer, returns pointer to the created customer.
- admin* registerAdmin(string name, string pass, int ID)
  → creates and registers a new admin, returns pointer to the created admin.

User Login:

- user* login(string name, string password, int ID)
  → verifies login credentials and returns pointer to the corresponding user if found.

File Operations (Persistence):

- `void saveusers() const`
  `→ saves user data to storage.`
- `void loadingusers()`
  `→ loads user data from storage.`

main function:

`void handlemenu2()`
` to show the main menu and this will be the function that gets called in the main function to compile the program`

Testing and Verification

Testing Objectives:

- verify accurate rental duration tracking.
- Confirm file persistence between program sessions like saving the time and the bikes data
- Test input validation and error handling
- Ensure rental cost calculation is correct
- Test registration and logging

Test Methods:

- Manual testing of each menu option like adding and deleting and login and registering and exiting and renting and returning . while checking the data is preserved on the bike_data file
- Running program, closing it, and checking data of the bikes and the data of the customers and their rented bikes .
- Providing invalid inputs to verify error handling to make sure the program will not crash
- Registering some customers while checking their data is preserved on the file

Results:

- All the functions are performed as expected.
- File persistence was successful for both the bike data and customers data and tracking the time is successful. And tracking the active bikes of the customers is also successful.
- Invalid inputs were handled peacefully.

Improvements and Extensions

Possible Improvements:

- Adding late fee calculations if a bike is returned after a certain period of time
- Allowing customers to change passwords
- Adding a more robust error handling
- Creating a GUI or web-based interface for smoother interaction.
- Adding a filtering system where the customer can look for an available bikes by brand or type...

Potential Extensions:

- Making a payment system simulation.
- Tracking bike condition or maintenance history using the duration of adding and the number of mileages for each bike
- Implement booking in advance and reservation queueing.

Difficulties Encountered

Challenges

. Adding a user system which were not that hard , but it was challengeable where I had to manage many stuff like their rented bikes and saving it to the files in case of leaving the program without returning it.

- time tracking complexities especially in the offline time .

Solution:
I started a new timer whenever a rental began and added the new running time to the previously saved duration. When the program was shut down, I stored the shutdown time to a file. Upon restarting, I loaded that time and calculated the offline duration, adding it to the existing rental time when the customer returns the bike. This way, even if the program was closed, rental durations remained accurate.

- persistent data storage especially saving the data the rented bike and their time and also saving all the active bikes and linking them to the customers data , to be able to return it again

Solution:
I used **virtual functions** in the base class, allowing me to call functions like getRentedBikeIDs() polymorphically. This let the program access customer-specific data (like which bikes a customer had rented) through a base class pointer or reference. It was crucial for keeping track of rentals and displaying the correct data for each logged-in customer.

- Error handling: Preventing invalid inputs without crashing the program.

Solution:
I added input validation loops. Whenever the program detected invalid input, it prompted the user to enter the value again, rather than exiting or proceeding with bad data. This made the system much more robust and user-friendly.

Conclusion

In this project, I successfully designed and implemented a bike rental management system that not only handled core features like customer registration, bike rental, and returns, but also managed to save all data and accurately track rental times across program sessions — including offline periods, to simulate a real service. Through the use of virtual functions and file handling, the program effectively linked customers to their rentals, saving the data to files and loading it again when needed. Overall, this project greatly improved my skills in object-oriented programming, especially working with objects and managing the data stored within them. It also strengthened my experience with the STL library, where I applied various useful tools and techniques.