## LIBRARY MANAGEMENT SYSTEM

SRM University – AP, Andhra Pradesh

Bachelor of Technology/Master of Technology

In

Computer Science and Engineering School of Engineering and Sciences



Under the Guidance of

#### (Karnena Kavitha Rani)

Submitted by

|  |  |  |
| --- | --- | --- |
| NAME | REG NO | BRANCH & CLASS |
| BHAVISHYA | AP23110010266 | Btech CSE E |
| YAGNA SRI | AP23110010297 | Btech CSE E |
| SIVA NIKSHYA | AP23110010 | Btech CSE E |

###### SRM University–AP Neerukonda, Mangalagiri, Guntur Andhra Pradesh –522240

###### [November,**2024**]

**Certificate**

Date: 16-Nov-24

This is to certify that the work present in this Project entitled **“Library Management System”** has been carried out by our team and our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

Prof. Kavitha Rani Karnena

Assistance Professor.

Department of Computer Science.

# Acknowledgements

###### This section acknowledges the contributors, tools, and resources that were essential to the development of the Library Management System.

###### We thank our development team, testers, and reviewers for their dedication. Gratitude is also extended to tools like C++ libraries, IDEs (e.g., Visual Studio), and Git for facilitating development. Resources such as Stack Overflow and Cplusplus.com provided invaluable support in overcoming technical challenges.

**Highlights of Functionality**

**1. Catalog Management:** Organizes and manages books, journals, and other resources in the library.

**2. User Management:** Handles user accounts, including member registration, profiles, and borrowing privileges.

**3. Circulation Control:** Manages book checkouts, returns, reservations, and overdue notifications.

**4. Inventory Tracking:** Tracks available stock, damaged items, and lost books in real time.

**5. Search & Discovery:** Provides powerful search options for locating books by title, author, genre, or keywords.

**6. Fines & Fees Management:** Automates fine calculations for overdue returns and manages payments.

**7. Reporting & Analytics:** Generates detailed reports on user activity, borrowing trends, and inventory status.

**8. Advanced Features:** Includes integration with digital resources, personalized recommendations, and accessibility features for users with special needs.

# Table of Contents

Certificate

[Acknowledgements](#_Toc182476934)

[Table of Contents](#_Toc182476935)

[Abstract](#_Toc182476936)

[List of figures](#_Toc182476938)

* 1. Introduction

1. Inheritance and Core Classes
2. Handling Library Requests
3. Library Operation and Maintenance
4. Error Handling
5. User Interaction
6. Data Storage and Management
   1. Methodology
      1. Requirements Analysis
      2. System Design
      3. Implementation
      4. Testing
      5. Enhancements
      6. User Documentation and Deployment
   2. Discussion
   3. Concluding Remarks
      1. Data Persistence
      2. Advanced Features
      3. Scalability
   4. Future Work
   5. Code
   6. Reference
   7. Input and output

# ABSTRACT :

# The Library Management System comprises the following components:

# Book Class: Manages details of books, including title, author, genre, availability status, and unique identifiers.

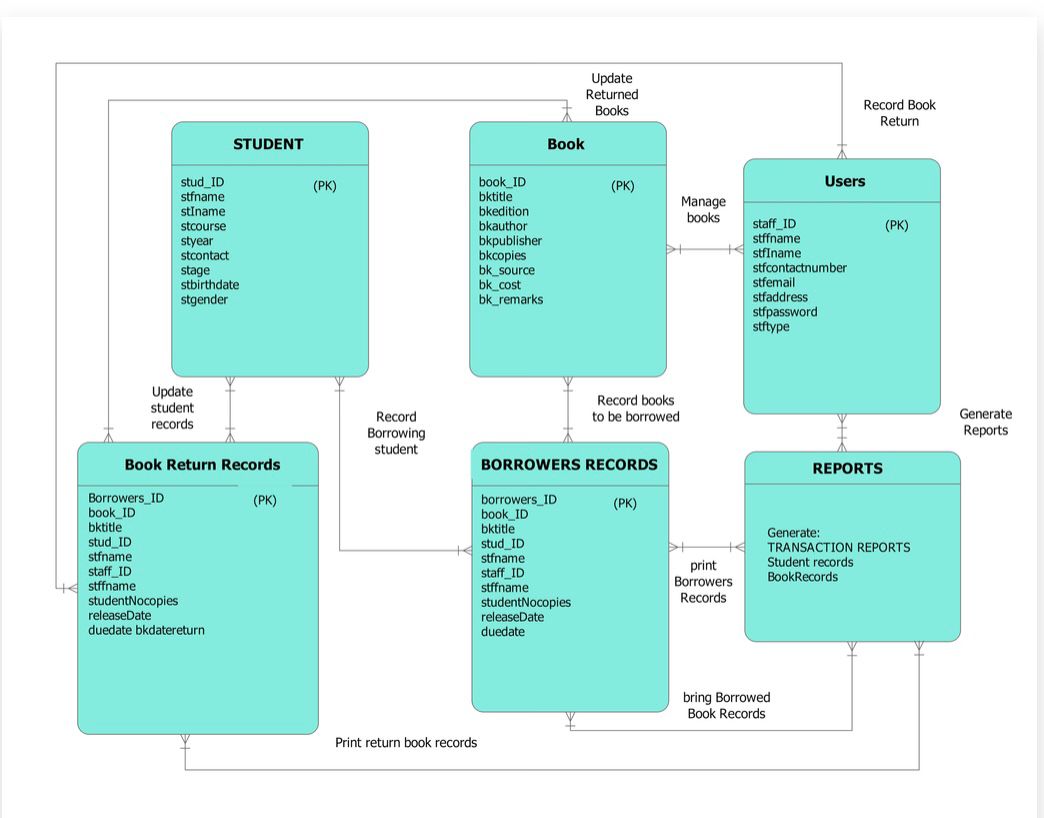
# Member Class: Handles user profiles, borrowing history, active loans, and membership details.

# Transaction Class: Represents each borrowing, return, or reservation transaction with book details, member information, and timestamps.

# Library Controller Class: Oversees system functions such as book checkouts, returns, inventory management, and search operations, ensuring efficient library operations.

# LMS Exception Class: Handles errors like overdue fines, unavailable resources, and invalid transactions to maintain system reliability.

# List of figures :



**1. Introduction**

The Library Management System (LMS) is a robust solution designed to optimize library operations. It simplifies resource management, enhances user experience, and reduces administrative workload. The system helps librarians and administrators manage books, track user activity, and maintain inventory effectively.

Developed using C++, the LMS is modular, efficient, and easy to maintain. It supports book borrowing, reservations, overdue notifications, and report generation. With intuitive error handling and a user-friendly interface, the system ensures smooth and reliable operations, making it suitable for libraries of all sizes.

**a) Inheritance and Core Classes**

Inheritance is a key feature used in the LMS to represent essential library components. Core classes include Book, Member, Transaction, and Library Controller.

***Base Class:*** Entity

The `Entity` base class includes common attributes such as `id` and `name`, shared by various components (e.g., books and members). It also features a pure virtual function, `display()`, requiring subclasses to define specific display behaviors.

**b) Book and Member Classes**

***Book Class* :**

Represents each book in the library, with attributes like title, author, ISBN, availability status, and location. Functions include `checkAvailability()` and `borrowBook()`.

***Member Class*  :**

Represents library members, tracking their details such as name, membership ID, borrowing history, and active loans. Functions include `borrow()` and `returnBook()`.

**c) Transaction Management**

The Transaction Class manages borrowing and returning activities. Each transaction logs details such as book ID, member ID, timestamps, and due dates. The class includes a `show()` function to display transaction details and ensures overdue fines are calculated automatically.

**d) Error Handling**

The LMS Exception Class inherits from `std::runtime\_error` to handle library-specific errors, such as unavailable books, invalid member IDs, and overdue penalties. This custom exception provides clear error messages to assist users in resolving issues.

**e) User Interaction**

The LMS features a console-based interface for easy interaction. The `main()` function provides a menu-driven system, allowing users to:

- Add new books or members.

- Borrow, return, or reserve books.

- Generate reports on library activity.

- Check book availability and member details.

f) Data Storage and Management

The LMS uses `vector` containers (e.g., `vector<Book>`, `vector<Member>`, `vector<Transaction>`) to store and manage data. This design ensures efficient handling of additions, deletions, and updates, making the system adaptable and scalable.

This C++ implementation demonstrates the language's efficiency in managing real-time operations, offering a reliable and effective Library Management System for libraries with diverse needs.

**2.Methodology**

**`a) Requirements Analysis**

The first step in designing a Library Management System involved understanding the requirements critical to its operation. The focus was on essential tasks such as managing book inventory, handling member registrations, processing book checkouts and returns, and tracking overdue items. Additionally, the system needed to be user-friendly, enabling librarians and staff to efficiently perform tasks such as cataloging books, searching the database, generating reports, and managing fines. Other priorities included providing features for tracking book reservations, maintaining member history, and integrating with digital resources where applicable.

The aim was to create a system that not only streamlined library operations but also improved the overall user experience for both staff and library patrons.

**b)System Design**

**Object-Oriented Design:**

* To represent real-world components and relationships in a Library Management System (LMS), the system was built using Object-Oriented Programming (OOP) design principles. The key elements of the system were:
  + Book, Member, Transaction, and Staff
  + Each of these elements was represented by a class, with specific methods to handle the data and operations relevant to that component. For example:
* **Book Class**: Stored details such as title, author, ISBN, publication year, and availability status. Methods included updating availability and retrieving book information.
* **Member Class**: Represented library members, with attributes like member ID, name, contact details, and borrowed books. Methods handled member registration, updating contact details, and managing borrowing history.

* **Transaction Class** : Processed operations such as book checkouts, returns, and fines, linking books with members and timestamps.

* **Staff Class**: Managed administrative functions, including adding books to the inventory, generating reports, and resolving member queries.

**Inheritance:**

Shared features across classes in the Library Management System (LMS) were implemented through inheritance. For instance**:**

**-** Both the `Book` and `Member` classes inherited common attributes (like `id`, `name`, and `dateAdded`) from a base class called `Entity`.

**-** Base Class (`Entity`): Provided common functionality, such as generating unique IDs and storing creation timestamps**.**

**- Derived Classes:**

- The `Book` class extended the `Entity` class to include attributes like `author`, `ISBN`, and `availabilityStatus`.

**-** The `Member` class extended the `Entity` class to include attributes like `membershipType`, `contactDetails`, and `borrowedBooks**`.**

This use of inheritance reduced code duplication, promoted reusability, and ensured consistent implementation of shared attributes and methods across related classes. It also provided a clear hierarchy that reflected the relationships between components of the library system.

**Encapsulation:**

In the Library Management System (LMS), each class encapsulated its data, ensuring that implementation details were hidden from users and other components. This design allowed secure management of data while providing well-defined interfaces for interaction.

For example:

**- Book Class:**

**-** Encapsulated attributes like `title`, `author`, `ISBN`, and `availabilityStatus`.

- Provided public methods like `checkAvailability()` and `updateStatus()` to access and modify these attributes, without directly exposing the underlying data.

**- Member Class:**

**-** Encapsulated member-specific details such as `memberID`, `name`, `borrowedBooks`, and `contactDetails`.

- Offered methods like `borrowBook(bookID)` and `returnBook(bookID)` to manage borrowing and returning operations**.**

**- Transaction Class :**

**-** Encapsulated attributes such as `transactionID`, `memberID`, `bookID`, `issueDate`, and `dueDate`.

- Provided methods to calculate fines or update transaction status.

**Polymorphism:**

In the Library Management System (LMS), polymorphism was implemented using virtual functions, enabling different types of `Entity` objects (such as `Book` and `Member`) to perform actions or display information uniquely based on their specific roles.

**For example:**

**Base Class (`Entity`):**

Provided a virtual method, `displayDetails()`, which could be overridden by derived classes**.**

**Derived Classes:**

**Book Class:** Implemented `displayDetails()` to show attributes like title, author, ISBN, and availability.

**Member Class:** Overrode `displayDetails()` to display member-specific details such as name, contact information, and borrowed books.

**c)** **Implementation**

* **C++ Programming:**

The LMS was developed in C++ to provide a strong and scalable structure, using features like classes, vectors (for flexible data storage), and exception handling.

The system used vectors to store data about books, members, and transactions, enabling efficient management and easy expansion as the library's collection and user base grew.

**User Interface:**

The LMS featured a straightforward, menu-driven console interface for user interaction. Through this interface, users could input data, select tasks, and receive system feedback. Input validation ensured that user-provided data, like books, memebers and transactions, was correct before processing.

**d) Testing**

**Unit Testing:**

Each function was tested independently to ensure it worked correctly. For instance, tests confirmed that book requests were logged accurately, members responded as expected, and transactions records were updated properly.

**Integration Testing:**

Once individual components were tested, the system was tested as a whole to verify that modules (such as request handling, maintenance scheduling, and book movement tracking) interacted smoothly.

**Edge Case Testing:**

The system was thoroughly tested for unusual scenarios, such as invalid book IDs, incorrect member details, or improperly formatted data entries, to ensure it handled these cases gracefully without errors or crashes.

**e) Enhancements**

**Efficiency:**

By using vector containers, the system could handle an increasing volume of data with minimal impact on performance. The modular structure of each class and function also made it easier to add features and optimize performance as needed.

**Code Refactoring:**

Throughout development, the code was reviewed and refined for clarity and efficiency. Tasks were broken down into simpler methods, and repetitive code was consolidated into functions, improving readability and reusability.

**f) User Documentation and Deployment**

**Deployment:**

After extensive testing, the LMS was launched as a console-based application for use by building operators and maintenance staff.

**User Documentation:**

A user manual was provided, covering essential **Library Management System (LMS)** functions, such as adding and managing books, registering and updating member information, processing book checkouts and returns, and generating reports on library operations.

These development phases resulted in a **Library Management System** that is comprehensive, scalable, and user-friendly, effectively supporting real-time library operations and providing a seamless experience for staff and patrons alike.

**3.Discussion**

 The **Library Management System (LMS)** built in C++ offers a robust solution for managing essential library operations, such as handling book checkouts and returns, tracking member activity, managing inventory, and generating reports.

 Object-oriented programming principles drive the system’s design, ensuring it is modular, reusable, and scalable.

 **Dynamic memory management** is achieved through vectors, enabling efficient storage and organization of data as the collection of books and member records grows.

 **Transaction management** supports seamless operation by linking members, books, and due dates, while tracking overdue items and fines ensures accountability.

 Key operations, such as book inventory updates, member registration, and transaction handling, are managed through specific functions to ensure smooth and reliable performance.

 The console-based interface is functional and interactive, offering users clear options for managing the library. However, a graphical interface could enhance usability by providing a more modern and intuitive experience.

 Basic exception handling through the LMS Exception class enables meaningful error reporting and ensures the system can handle invalid inputs, such as incorrect book IDs or member details, without crashing.

 Improvements could include refining data structures for faster processing of transactions, expanding functionality for member management and book requests, introducing predictive analytics for book demand, creating a graphical user interface (GUI) for easier interactions, and enhancing error handling to ensure smoother operations.

 More advanced exception handling could cover a broader range of issues, such as requests for non-existent books, invalid member information, or conflicts in reservation requests.

 With scalability in mind, future upgrades to the LMS are necessary to handle larger-scale operations, especially as the number of books and members grows, and more complex transactions need to be managed.

 The system effectively handles core library functions, yet opportunities exist for further enhancements, such as automating book recommendations, supporting digital resource management, or implementing an integrated library system (ILS).

 Advanced features like intelligent scheduling for book reservations, real-time monitoring of inventory, tracking of overdue books and fines, remote notifications for library staff, and advanced reporting tools could make the LMS more versatile and comprehensive.

 Implementing user roles and access controls would address security concerns, ensuring that only authorized staff can adjust settings, add new books, modify member records, or access sensitive data such as financial reports or overdue accounts.

1. **Concluding Remarks**

The Library Management System (LMS) built in C++ provides an effective and straightforward approach to managing key library functions. By leveraging object-oriented programming principles such as inheritance, polymorphism, and encapsulation, the system is structured to handle library operations in a modular, organized, and scalable way. This foundation effectively supports real-world library operations by managing book checkouts, member registrations, inventory, and transaction tracking.

The system uses vectors to manage dynamic data storage, allowing it to expand easily as the number of books, members, and transactions grows. The Transaction and Member classes simplify the process of handling book checkouts, returns, and fines, ensuring smooth interaction between system components. With input validation and simple exception handling, the system is also resilient, ensuring errors such as invalid book IDs, member details, or transaction dates are detected and managed efficiently.

1. **Data Persistence:**

The system currently lacks permanent data storage, which could be improved by integrating a database or file-based storage. This enhancement would allow the system to retain data even after closing, making it suitable for continuous, long-term use.

1. **Advanced Features:**

The **Library Management System (LMS)** could be expanded with advanced capabilities, such as **predictive scheduling** for book reservations based on demand trends, **remote monitoring** of book availability, **energy efficiency tracking** for library facilities, and **real-time maintenance alerts** for library equipment .

1. **Scalability:**

While the current design works well for small to medium-sized libraries, managing larger volumes of data, such as a large collection of books or a high number of patrons, may require more efficient data structures, like databases or hash maps, to maintain optimal performance.

In summary, the Library Management System (LMS) provides a strong foundation for managing library operations. It showcases how core C++ concepts can be used to build a flexible and functional solution, with significant potential for future development to meet the needs of larger libraries or more complex library management scenarios. The current implementation offers a reliable platform for future growth, with numerous opportunities to add features such as automated book recommendations, integrated digital resource management, and advanced reporting capabilities to support increasingly sophisticated library systems.

1. **Future Work**

The current Library Management System (LMS) in C++ provides essential functionalities for managing library operations, such as handling book checkouts and returns, tracking member activity, managing inventory, and generating transaction reports.

Looking ahead, potential improvements include integrating a databasefor permanent data storage and faster data retrieval, which would significantly improve scalability and performance as the library grows. A Graphical User Interface (GUI) would also enhance user experience, making it easier for staff to interact with the system for tasks like cataloging books, managing transactions, and running reports.

Security could be enhanced through user authentication and role-based access controls, ensuring that only authorized personnel can access sensitive data or change critical system settings. This would prevent unauthorized access and ensure data integrity.

The system could also benefit from advanced features such as smart scheduling for book reservations, load balancing for book checkout management, and predictive maintenance notifications for library equipment. For example, predictive maintenance could be implemented for self-checkout machines or library kiosks, allowing staff to anticipate and address issues before they disrupt service.

Further improvements could include report generation for operational insights, such as tracking overdue books, library usage patterns, and resource allocation. Additionally, incorporating energy efficiency tracking for library systems (e.g., lighting, heating, or cooling) could promote sustainability.

Extending the LMS with a mobile app for both staff and patrons would further enhance the library experience, offering features like mobile book reservations, notifications for overdue items, and access to library catalogs on the go.

By incorporating these enhancements, the Library Management System would become a more comprehensive, user-friendly, and future-proof solution for managing modern libraries. support, multilingual capabilities, and cloud integration would provide remote access, improved scalability, and secure data backup.

With these enhancements, the LMS could evolve into a powerful and comprehensive system capable of managing elevator operations in large buildings or complexes. It would offer greater reliability and efficiency for everyday operations.

These upgrades would make the LMS more user-friendly, scalable, and advanced, benefiting building operators, maintenance teams, and passengers by optimizing lift performance, reducing downtime, and improving the overall experience of managing elevators.

**CODE FOR LIBRARY MANAGEMENT SYSTEM**

#include <iostream>

#include <string>

#include <vector>

#include <algorithm> // For find()

using namespace std;

class Book {

public:

string title;

string author;

bool isBorrowed;

Book(string t, string a) : title(t), author(a), isBorrowed(false) {}

void display() const {

cout << "Title: " << title << ", Author: " << author;

if (isBorrowed) {

cout << " (Borrowed)" << endl;

} else {

cout << " (Available)" << endl;

}

}

};

class User {

public:

string studentRegNo; // Student registration number

string name;

vector<string> borrowedBooks;

User(string regNo, string n) : studentRegNo(regNo), name(n) {}

void display() const {

cout << "Student Reg No: " << studentRegNo << ", Name: " << name << ", Borrowed Books: ";

if (borrowedBooks.empty()) {

cout << "None" << endl;

} else {

for (const string& bookTitle : borrowedBooks) {

cout << bookTitle << " ";

}

cout << endl;

}

}

bool hasBorrowed(const string& bookTitle) const {

return find(borrowedBooks.begin(), borrowedBooks.end(), bookTitle) != borrowedBooks.end();

}

};

class Library {

vector<Book> books;

vector<User> users;

Book\* findBook(const string& title) {

for (auto &book : books) {

if (book.title == title) {

return &book;

}

}

return nullptr;

}

User\* findUser(const string& studentRegNo) {

for (auto &user : users) {

if (user.studentRegNo == studentRegNo) {

return &user;

}

}

return nullptr;

}

public:

void addBook(const string& title, const string& author) {

if (findBook(title) == nullptr) {

books.emplace\_back(title, author);

cout << "Book added: " << title << " by " << author << endl;

} else {

cout << "Error: A book with title \"" << title << "\" already exists." << endl;

}

}

void addUser(const string& studentRegNo, const string& name) {

if (findUser(studentRegNo) == nullptr) {

users.emplace\_back(studentRegNo, name);

cout << "User added: " << name << endl;

} else {

cout << "Error: A user with registration number \"" << studentRegNo << "\" already exists." << endl;

}

}

void borrowBook(const string& studentRegNo, const string& title) {

Book\* book = findBook(title);

User\* user = findUser(studentRegNo);

if (book && user) {

if (book->isBorrowed) {

cout << "Sorry, the book \"" << title << "\" is not available. It has already been borrowed." << endl;

} else if (user->hasBorrowed(title)) {

cout << "You have already borrowed the book \"" << title << "\"." << endl;

} else {

book->isBorrowed = true;

user->borrowedBooks.push\_back(title);

cout << "Book borrowed: " << title << " by " << user->name << endl;

}

} else {

if (!book) cout << "Error: Book not found." << endl;

if (!user) cout << "Error: User not found." << endl;

}

}

void returnBook(const string& studentRegNo, const string& title) {

Book\* book = findBook(title);

User\* user = findUser(studentRegNo);

if (book && user) {

if (book->isBorrowed) {

auto it = find(user->borrowedBooks.begin(), user->borrowedBooks.end(), title);

if (it != user->borrowedBooks.end()) {

book->isBorrowed = false;

user->borrowedBooks.erase(it);

cout << "Book returned: " << title << " by " << user->name << endl;

} else {

cout << "Error: You did not borrow this book." << endl;

}

} else {

cout << "Error: Book was not borrowed." << endl;

}

} else {

if (!book) cout << "Error: Book not found." << endl;

if (!user) cout << "Error: User not found." << endl;

}

}

void searchBook(const string& title) const {

bool found = false;

for (const auto &book : books) {

if (book.title.find(title) != string::npos) {

book.display();

found = true;

}

}

if (!found) {

cout << "No book found with title containing: " << title << endl;

}

}

void displayAllBooks() const {

cout << "Books in library:" << endl;

for (const auto &book : books) {

book.display();

}

}

void displayAllUsers() const {

cout << "Registered users:" << endl;

for (const auto &user : users) {

user.display();

}

}

};

int main() {

Library library;

int numBooks, numUsers;

// Input number of books and users

cout << "Enter the number of books: ";

cin >> numBooks;

cin.ignore(); // To clear the newline after the number input

// Add books

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

string title, author;

cout << "Enter details for book " << i + 1 << ":\n";

cout << "Title: ";

getline(cin, title);

cout << "Author: ";

getline(cin, author);

library.addBook(title, author);

}

// Input number of users

cout << "Enter the number of users: ";

cin >> numUsers;

cin.ignore(); // To clear the newline after the number input

// Add users

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

string studentRegNo, name;

cout << "Enter details for user " << i + 1 << ":\n";

cout << "Student Registration Number: ";

getline(cin, studentRegNo);

cout << "Name: ";

getline(cin, name);

library.addUser(studentRegNo, name);

}

// Display books and users

library.displayAllBooks();

library.displayAllUsers();

// Example of borrowing and returning books

string title, studentRegNo;

cout << "Enter student registration number to borrow a book: ";

getline(cin, studentRegNo);

cout << "Enter book title to borrow: ";

getline(cin, title);

library.borrowBook(studentRegNo, title);

// Trying to borrow the same book again

cout << "Enter student registration number to borrow a book again: ";

getline(cin, studentRegNo);

cout << "Enter book title to borrow again: ";

getline(cin, title);

library.borrowBook(studentRegNo, title); // This should give feedback that the book is not available

cout << "Enter student registration number to return a book: ";

getline(cin, studentRegNo);

cout << "Enter book title to return: ";

getline(cin, title);

library.returnBook(studentRegNo, title);

// Search for a book by title

cout << "Enter a title to search for: ";

getline(cin, title);

library.searchBook(title);

    return 0;

}

**CODE EXPLANATION:**

Explanation of the Code

This program implements a basic Library Management System using C++ classes. It allows managing books, users, borrowing, and returning books. Here’s a breakdown of the code:

1. Classes

• Book:

• Represents a book in the library with attributes:

• id (integer): Unique identifier for the book.

• title (string): Title of the book.

• author (string): Author of the book.

• isBorrowed (boolean): Status of whether the book is borrowed or available.

• Methods:

• display(): Outputs the book’s details.

• User:

• Represents a user of the library with attributes:

• userId (integer): Unique identifier for the user.

• name (string): Name of the user.

• borrowedBooks (vector of integers): List of IDs of books borrowed by the user.

• Methods:

• display(): Outputs the user’s details.

• Library:

• Manages the books and users in the library using two vectors:

• books: Vector of Book objects.

• users: Vector of User objects.

• Methods:

• addBook: Adds a new book to the library.

• addUser: Registers a new user.

• borrowBook: Allows a user to borrow a book.

• returnBook: Allows a user to return a book.

• searchBook: Searches for books by title.

• displayAllBooks: Displays all books in the library.

• displayAllUsers: Displays all users in the library.

• Private helper methods:

• findBook: Locates a book by its ID.

• findUser: Locates a user by their ID.

2. main() Function

• Demonstrates the usage of the Library system:

• Adds books and users.

• Displays books and users.

• Borrows and returns books with error handling for invalid actions.

• Searches for books by title.

Features

1. Book Management: Add books, display all books, and search by title.

2. User Management: Add users and display all registered users.

3. Borrow/Return Books: Handles borrowing and returning books with validation.

4. Error Handling: Detects and reports errors, such as:

• Borrowing an already borrowed book.

• Returning a book not borrowed.

• Invalid book/user IDs.

**INPUT AND OUTPUT :**

