Project Report: Advanced Library Book Management System

BS English (6th Semester): End Term Project

Instructor: Maida Naveed

Submitted by: Roshane Shahbaz

Date: June 20th, 2025

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

ISLAMABAD, PAKISTAN

(This is a graded project)

Project Report: Advanced Library Book Management System

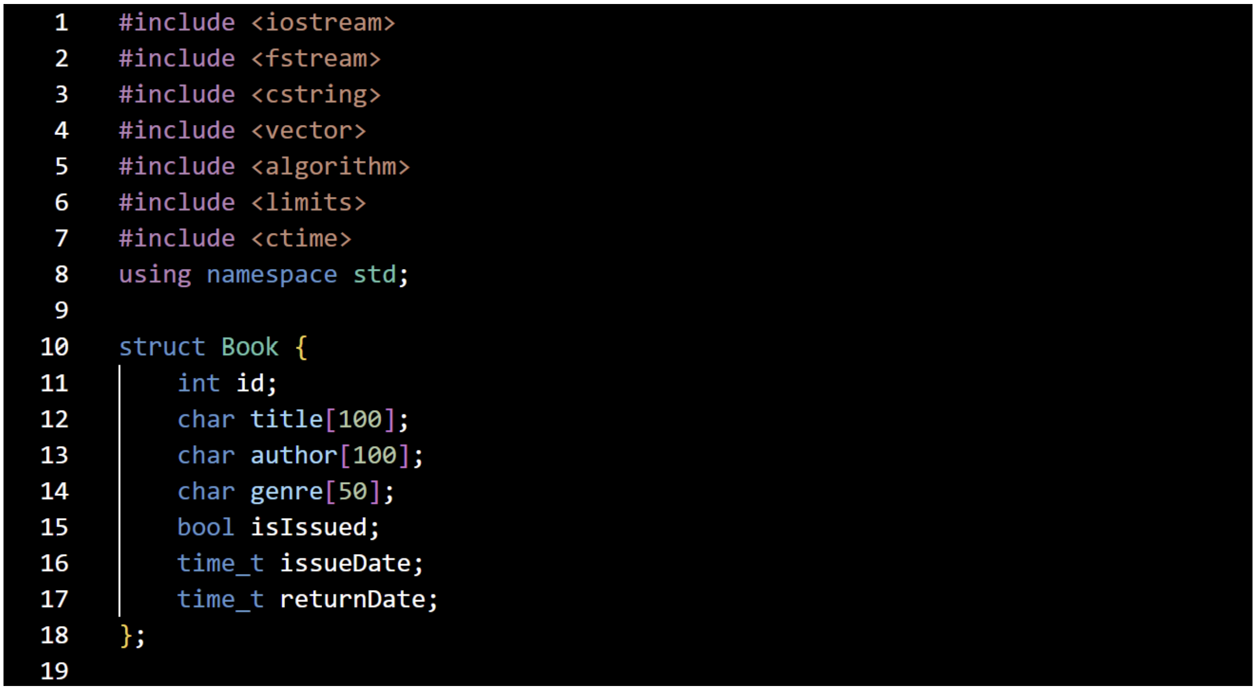
Developer: Roshane Shahbaz
Language: C++
Type: School Project
Interface: Console-based
Data Storage: Binary File (library.dat)

Introduction

This project implements a functional and interactive Library Management System using the C++ programming language. Designed as a school-level assignment, it simulates real-world library operations within a console-based interface. The system allows users to manage books through features such as adding new entries, issuing and returning books, deleting outdated records, searching the catalog, and generating summary reports. The application utilizes binary file handling for persistent data storage and incorporates modular programming, date-based logic, and input validation to ensure efficiency, reliability, and user-friendliness. This system not only demonstrates coding skills but also provides a practical understanding of how libraries manage data in backend systems.

Header Inclusions and Book Structure

The project begins by including the necessary standard C++ headers that provide functionality for input/output operations, file handling, string manipulation, dynamic data structures, and time processing. These libraries enable the system to store and process book records effectively while maintaining performance and structure. The Book structure is defined to represent each library record in memory and on disk.

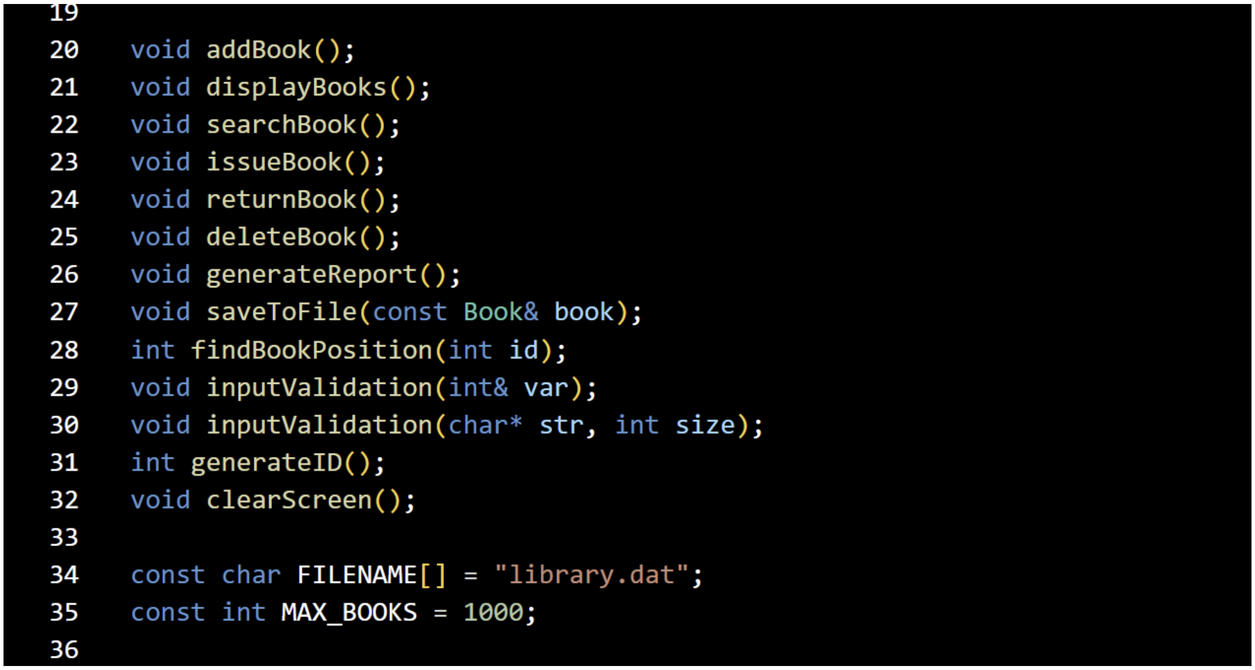


```cpp
#include <iostream>
#include <fstream>
#include <cstring>
#include <vector>
#include <algorithm>
#include <limits>
#include <ctime>
using namespace std;

struct Book {
    int id;
    char title[100];
    char author[100];
    char genre[50];
    bool isIssued;
    time_t issueDate;
    time_t returnDate;
};

```

This structure contains the essential metadata for every book entry. It holds the ID, title, author, genre, a flag indicating issuance status, and timestamps for when the book was issued and when it is due to be returned. Using a single, structured object to represent each book simplifies storage, search, and update operations throughout the program.

Function Prototypes and Constants

To maintain clarity and code organization, all function declarations are listed at the beginning of the program. This makes the source code easier to follow and avoids forward-reference errors during compilation.

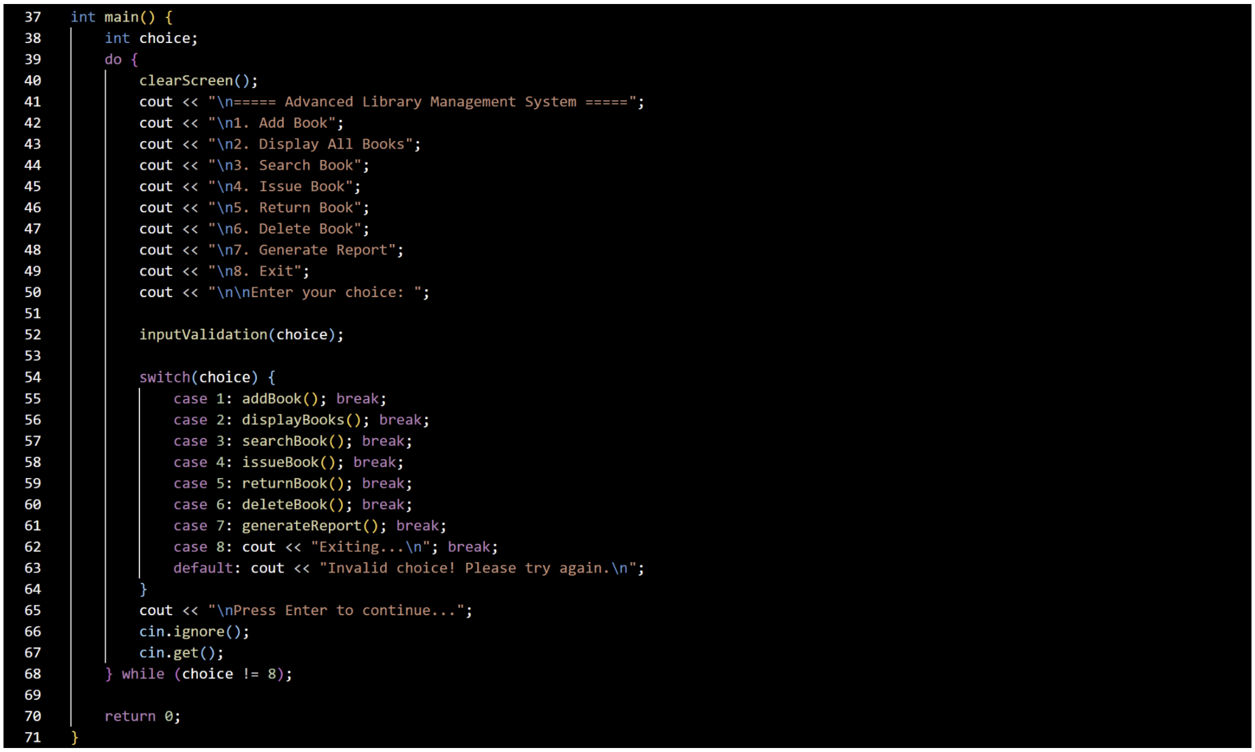


```
void addBook();
void displayBooks();
void searchBook();
void issueBook();
void returnBook();
void deleteBook();
void generateReport();
void saveToFile(const Book& book);
int findBookPosition(int id);
void inputValidation(int& var);
void inputValidation(char* str, int size);
int generateID();
void clearScreen();

const char FILENAME[] = "library.dat";
const int MAX_BOOKS = 1000;

```

Constants such as the file name used for data storage and the maximum number of books supported are defined globally. This makes it easy to update configurations from a single location if needed, and it promotes consistency across all functions that interact with the binary file.

Main Menu and Program Flow

The main part of the program is structured around a user-interactive menu that continuously prompts the user until they choose to exit. This menu includes all primary operations, such as adding, displaying, issuing, returning, deleting, and searching for books, as well as generating reports.

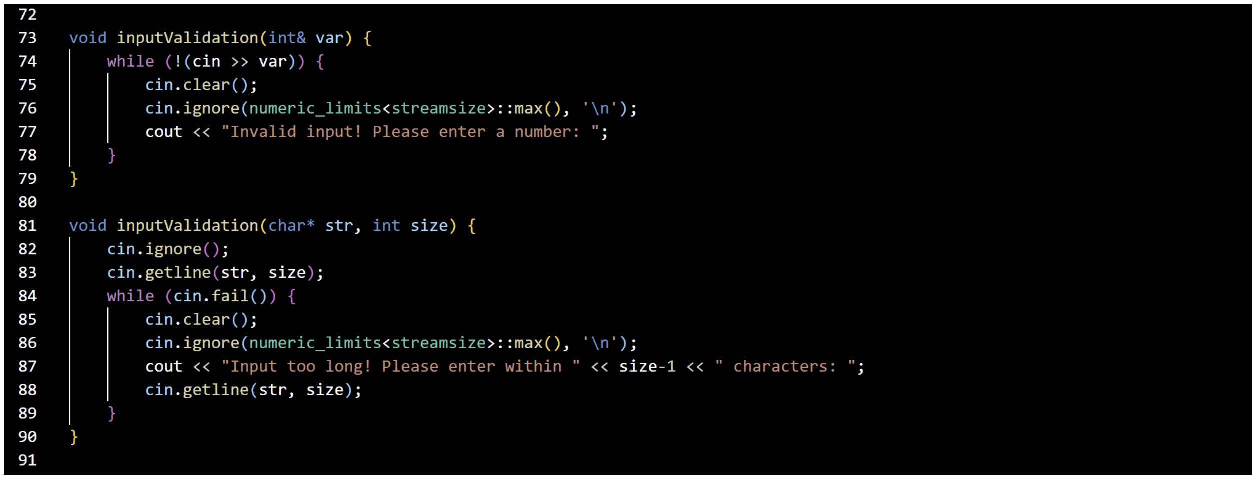


```cpp
int main() {
    int choice;
    do {
        clearScreen();
        cout << "\n===== Advanced Library Management System =====";
        cout << "\n1. Add Book";
        cout << "\n2. Display All Books";
        cout << "\n3. Search Book";
        cout << "\n4. Issue Book";
        cout << "\n5. Return Book";
        cout << "\n6. Delete Book";
        cout << "\n7. Generate Report";
        cout << "\n8. Exit";
        cout << "\n\nEnter your choice: ";

        inputValidation(choice);

        switch(choice) {
            case 1: addBook(); break;
            case 2: displayBooks(); break;
            case 3: searchBook(); break;
            case 4: issueBook(); break;
            case 5: returnBook(); break;
            case 6: deleteBook(); break;
            case 7: generateReport(); break;
            case 8: cout << "Exiting...\n"; break;
            default: cout << "Invalid choice! Please try again.\n";
        }
        cout << "\nPress Enter to continue...";
        cin.ignore();
        cin.get();
    } while (choice != 8);

    return 0;
}
```

A switch-case control structure is used to determine which function to execute based on the user’s selection. This design keeps the code organized and easy to navigate while giving users complete control over how they interact with the system.

Input Validation

Robust input validation functions are used throughout the program to handle both numeric and string inputs. These functions ensure that users enter appropriate values and prevent common issues like buffer overflows or type mismatches.

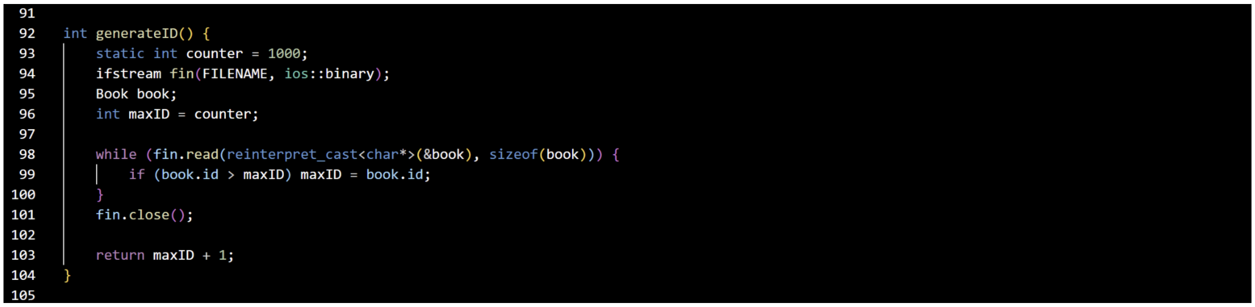


```cpp

void inputValidation(int& var) {
    while (!(cin >> var)) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Invalid input! Please enter a number: ";
    }
}

void inputValidation(char* str, int size) {
    cin.ignore();
    cin.getline(str, size);
    while (cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Input too long! Please enter within " << size-1 << " characters: ";
        cin.getline(str, size);
    }
}

```

By validating inputs before processing, the system maintains stability and prevents unexpected behavior or crashes. This is particularly important in console applications where the user can directly input data without constraints.

Auto-Generating Unique IDs

To ensure each book record has a unique identifier, the system automatically generates a new ID by scanning the binary file for the highest existing ID and incrementing it. This eliminates the risk of duplicate IDs and removes the need for users to enter an ID manually.

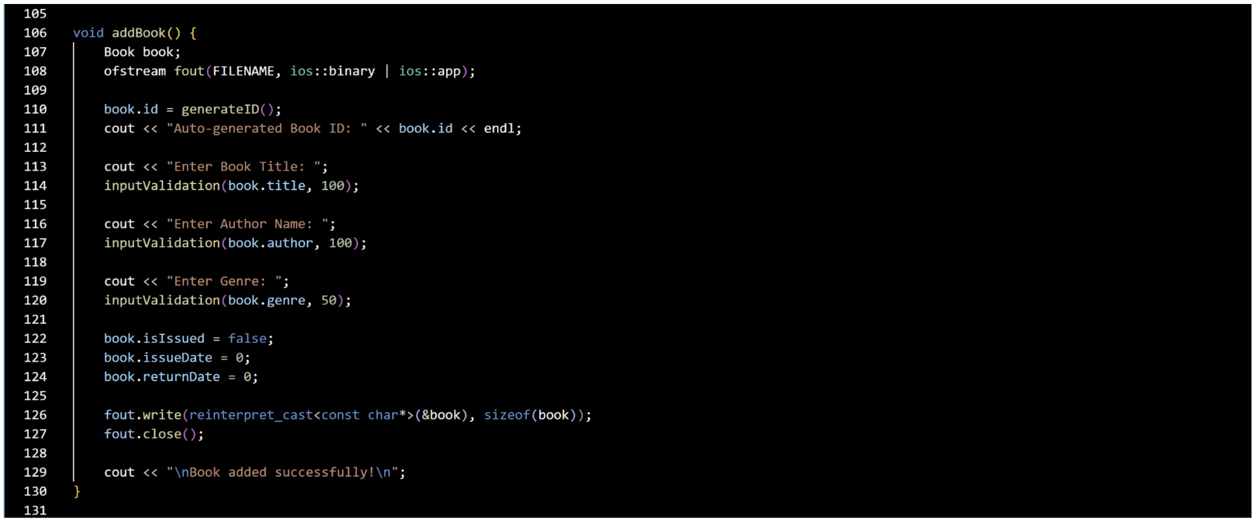


```cpp

int generateID() {
    static int counter = 1000;
    ifstream fin(FILENAME, ios::binary);
    Book book;
    int maxID = counter;

    while (fin.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        if (book.id > maxID) maxID = book.id;
    }
    fin.close();

    return maxID + 1;
}

```

This automated approach streamlines the book addition process and maintains consistent record-keeping practices within the system.

Adding Books

The function addBook() handles the input and file writing process. This function to add books collects information from the user, including the book's title, author, and genre. It then assigns a unique ID, initializes the issue and return status to default values, and stores the book in the binary file.

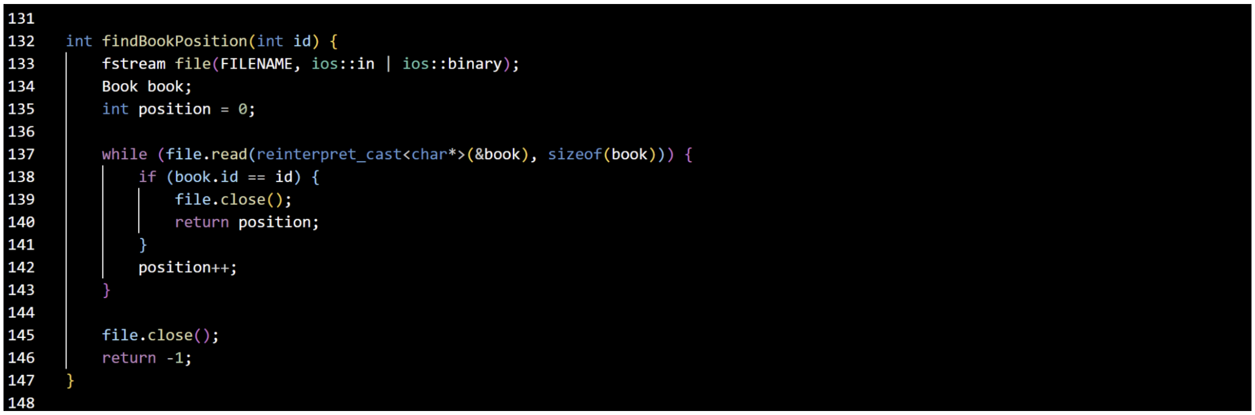


```cpp

void addBook() {
    Book book;
    ofstream fout(FILENAME, ios::binary | ios::app);

    book.id = generateID();
    cout << "Auto-generated Book ID: " << book.id << endl;

    cout << "Enter Book Title: ";
    inputValidation(book.title, 100);

    cout << "Enter Author Name: ";
    inputValidation(book.author, 100);

    cout << "Enter Genre: ";
    inputValidation(book.genre, 50);

    book.isIssued = false;
    book.issueDate = 0;
    book.returnDate = 0;

    fout.write(reinterpret_cast<const char*>(&book), sizeof(book));
    fout.close();

    cout << "\nBook added successfully!\n";
}

```

This process guarantees that all necessary information is stored for each book and that new records are immediately available for other operations like issuing or searching.

Finding Book Position

To update or remove a book, the program needs to locate its position in the binary file. This is done using a search function that reads through the file until the desired book ID is found, returning its position for further actions. To locate a book within the binary file, a helper function findBookPosition() is used.

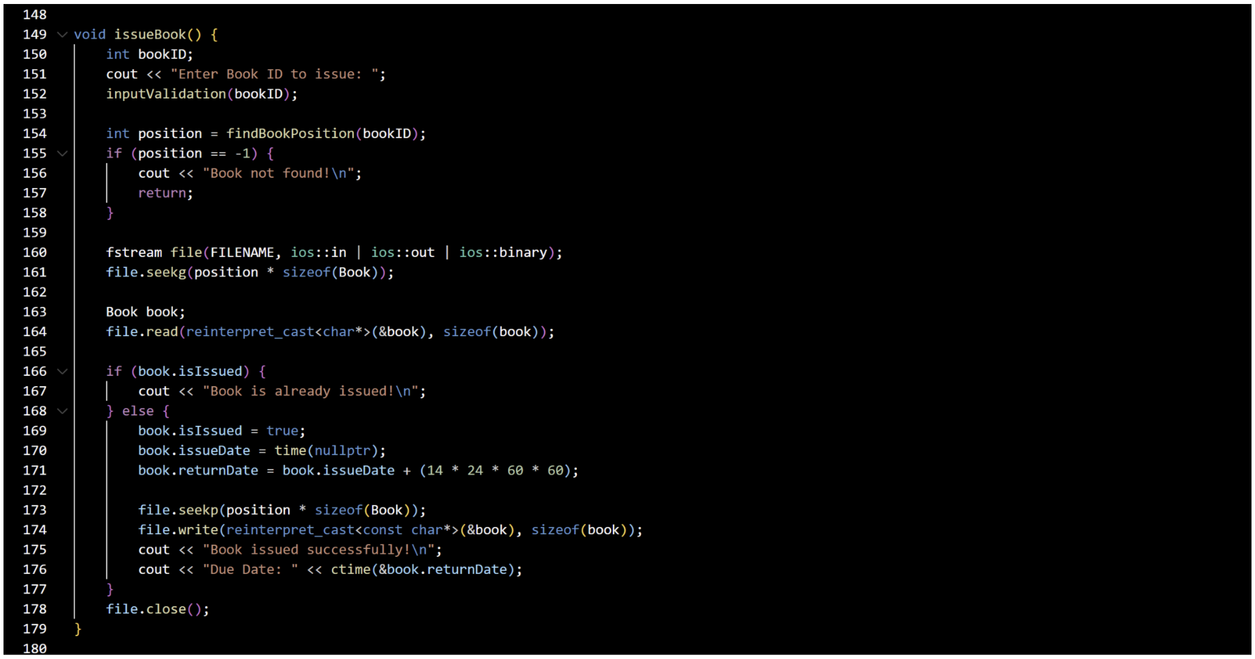


```cpp

int findBookPosition(int id) {
    fstream file(FILENAME, ios::in | ios::binary);
    Book book;
    int position = 0;

    while (file.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        if (book.id == id) {
            file.close();
            return position;
        }
        position++;
    }

    file.close();
    return -1;
}

```

This method allows the program to selectively update or overwrite a specific record, which is particularly useful for operations like issuing or returning books without rewriting the entire file.

Issuing Books

When a book is issued, the system updates its status and records the current date as the issue date. It also calculates a return date by adding a 14-day period to the issue date.

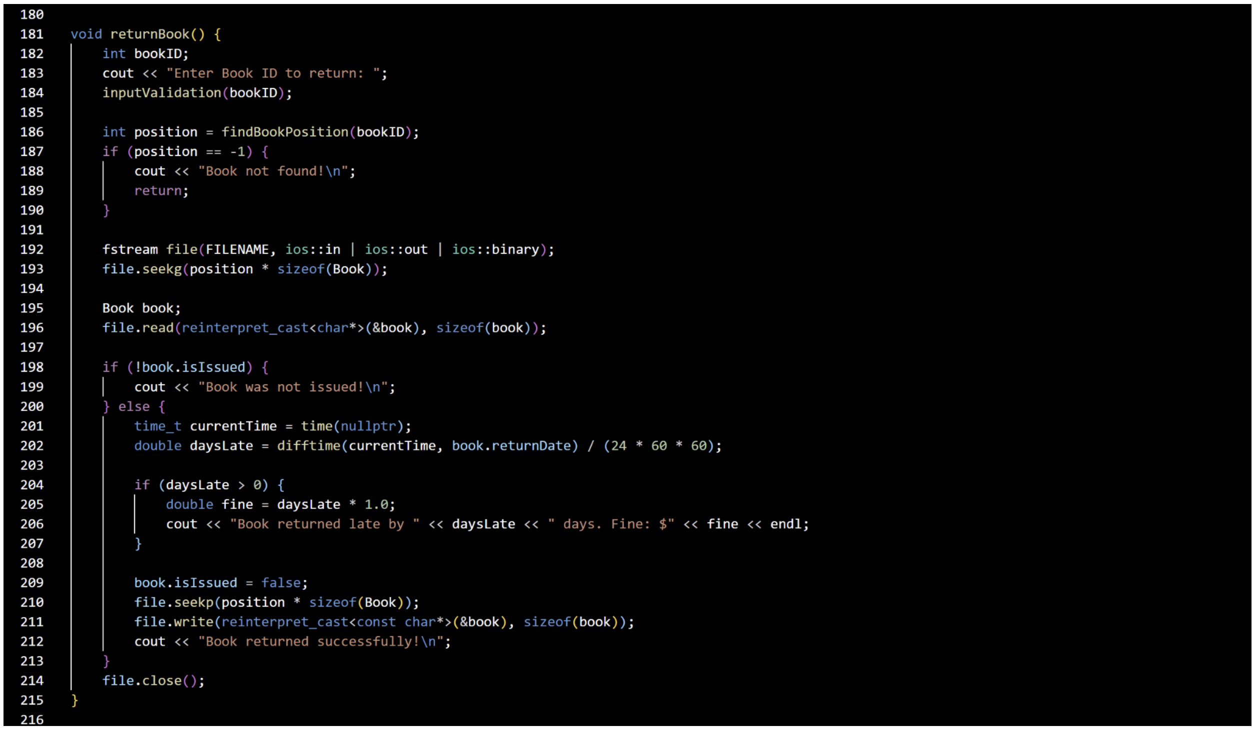


```cpp

void issueBook() {
    int bookID;
    cout << "Enter Book ID to issue: ";
    inputValidation(bookID);

    int position = findBookPosition(bookID);
    if (position == -1) {
        cout << "Book not found!\n";
        return;
    }

    fstream file(FILENAME, ios::in | ios::out | ios::binary);
    file.seekg(position * sizeof(Book));

    Book book;
    file.read(reinterpret_cast<char*>(&book), sizeof(book));

    if (book.isIssued) {
        cout << "Book is already issued!\n";
    } else {
        book.isIssued = true;
        book.issueDate = time(nullptr);
        book.returnDate = book.issueDate + (14 * 24 * 60 * 60);

        file.seekp(position * sizeof(Book));
        file.write(reinterpret_cast<const char*>(&book), sizeof(book));
        cout << "Book issued successfully!\n";
        cout << "Due Date: " << ctime(&book.returnDate);
    }
    file.close();
}

```

Here, ctime() is used to display the due date in a readable format. This function prevents a book from being issued if it is already checked out and provides a clear return timeline for the user. The calculated return date is displayed in a readable format to enhance the user experience.

Returning Books and Calculating Fines

Upon returning a book, the system checks if the current date exceeds the stored return date. If so, it calculates a fine based on the number of days the book is overdue and informs the user of the amount.

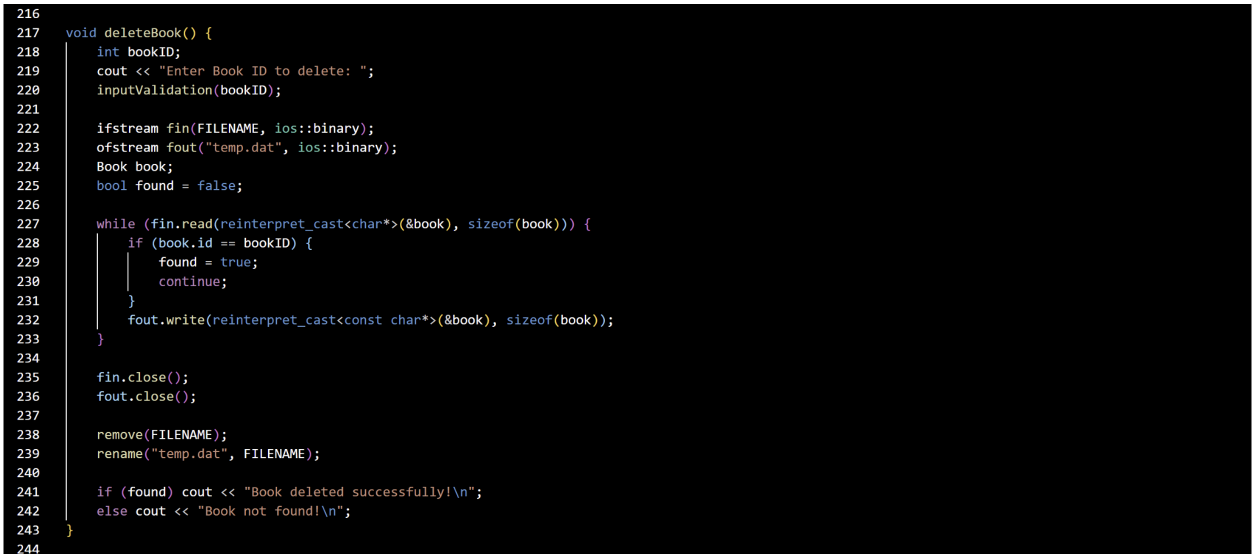


```cpp

void returnBook() {
    int bookID;
    cout << "Enter Book ID to return: ";
    inputValidation(bookID);

    int position = findBookPosition(bookID);
    if (position == -1) {
        cout << "Book not found!\n";
        return;
    }

    fstream file(FILENAME, ios::in | ios::out | ios::binary);
    file.seekg(position * sizeof(Book));

    Book book;
    file.read(reinterpret_cast<char*>(&book), sizeof(book));

    if (!book.isIssued) {
        cout << "Book was not issued!\n";
    } else {
        time_t currentTime = time(nullptr);
        double daysLate = difftime(currentTime, book.returnDate) / (24 * 60 * 60);

        if (daysLate > 0) {
            double fine = daysLate * 1.0;
            cout << "Book returned late by " << daysLate << " days. Fine: $" << fine << endl;
        }

        book.isIssued = false;
        file.seekp(position * sizeof(Book));
        file.write(reinterpret_cast<const char*>(&book), sizeof(book));
        cout << "Book returned successfully!\n";
    }
    file.close();
}

```

This feature introduces time-sensitive logic into the system, mirroring real-world library policies. It encourages timely returns and provides a practical use case for working with time functions in C++.

Deleting a Book

Books can be deleted from the catalog by copying all records except the one to be removed into a temporary file. The original file is then replaced by this filtered version.

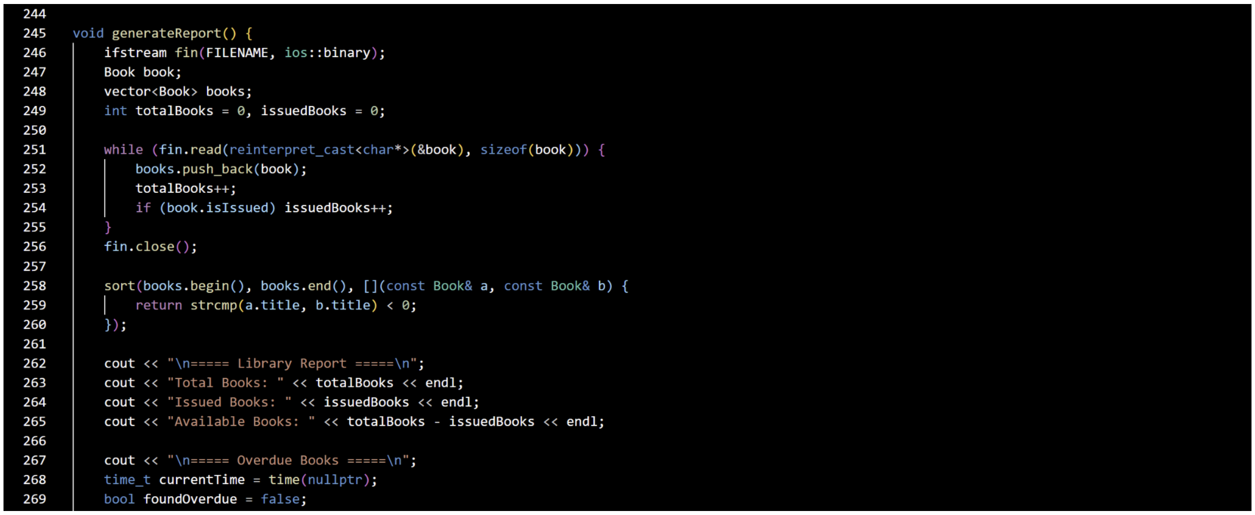


```cpp

void deleteBook() {
    int bookID;
    cout << "Enter Book ID to delete: ";
    inputValidation(bookID);

    ifstream fin(FILENAME, ios::binary);
    ofstream fout("temp.dat", ios::binary);
    Book book;
    bool found = false;

    while (fin.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        if (book.id == bookID) {
            found = true;
            continue;
        }
        fout.write(reinterpret_cast<const char*>(&book), sizeof(book));
    }

    fin.close();
    fout.close();

    remove(FILENAME);
    rename("temp.dat", FILENAME);

    if (found) cout << "Book deleted successfully!\n";
    else cout << "Book not found!\n";
}

```

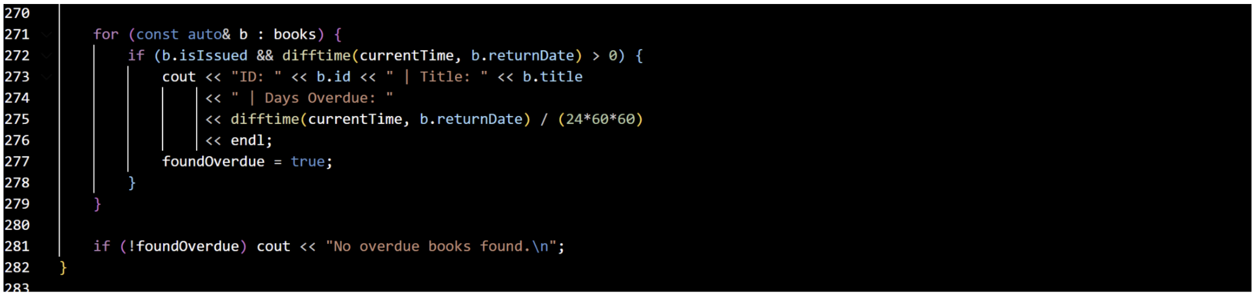
This approach effectively removes a book from the system without directly editing binary data, which is a limitation of most low-level file storage methods. It ensures data integrity while maintaining a clean and accurate file structure.

Generating Reports

The report generation function analyzes the entire book collection to count total books, identify how many are issued or available, and list any books that are overdue.



```cpp

void generateReport() {
    ifstream fin(FILENAME, ios::binary);
    Book book;
    vector<Book> books;
    int totalBooks = 0, issuedBooks = 0;

    while (fin.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        books.push_back(book);
        totalBooks++;
        if (book.isIssued) issuedBooks++;
    }
    fin.close();

    sort(books.begin(), books.end(), [](const Book& a, const Book& b) {
        return strcmp(a.title, b.title) < 0;
    });

    cout << "\n===== Library Report =====\n";
    cout << "Total Books: " << totalBooks << endl;
    cout << "Issued Books: " << issuedBooks << endl;
    cout << "Available Books: " << totalBooks - issuedBooks << endl;

    cout << "\n===== Overdue Books =====\n";
    time_t currentTime = time(nullptr);
    bool foundOverdue = false;
```

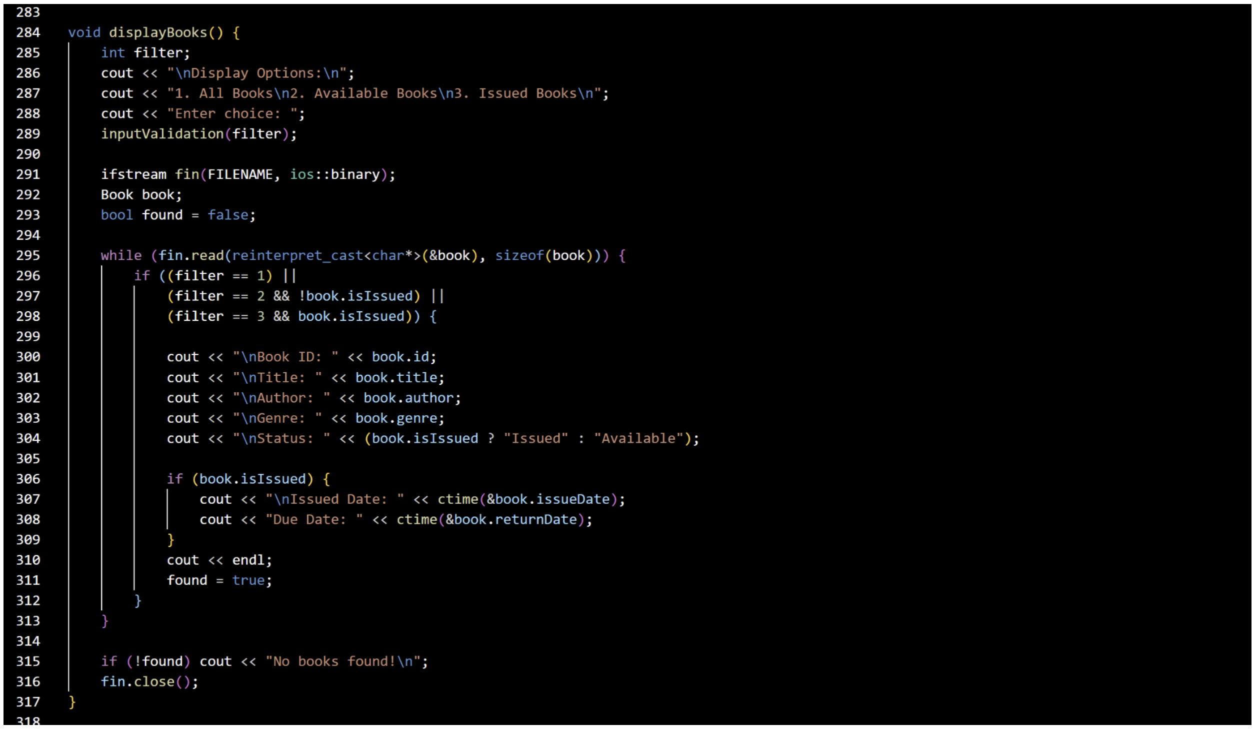


```
    for (const auto& b : books) {
        if (b.isIssued && difftime(currentTime, b.returnDate) > 0) {
            cout << "ID: " << b.id << " | Title: " << b.title
                 << " | Days Overdue: "
                 << difftime(currentTime, b.returnDate) / (24*60*60)
                 << endl;
            foundOverdue = true;
        }
    }

    if (!foundOverdue) cout << "No overdue books found.\n";
}
```

Books are temporarily stored in a dynamic container for sorting and processing, allowing the report to be displayed in a user-friendly and organized format. This feature is especially useful for library administrators to get a quick overview of inventory and user compliance.

Displaying Books

The system allows users to view all books or filter them based on availability or issuance status. Depending on the selected filter, it displays relevant details such as the book's ID, title, author, genre, and status.



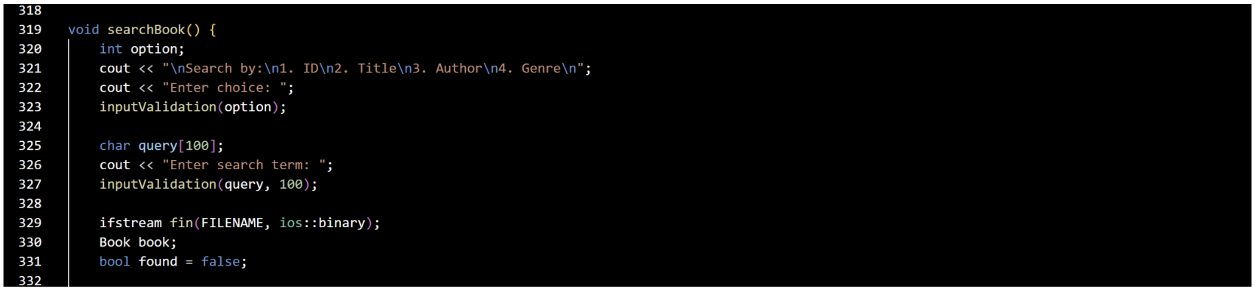
```

void displayBooks() {
    int filter;
    cout << "\nDisplay Options:\n";
    cout << "1. All Books\n2. Available Books\n3. Issued Books\n";
    cout << "Enter choice: ";
    inputValidation(filter);

    ifstream fin(FILENAME, ios::binary);
    Book book;
    bool found = false;

    while (fin.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        if ((filter == 1) ||
            (filter == 2 && !book.isIssued) ||
            (filter == 3 && book.isIssued)) {

            cout << "\nBook ID: " << book.id;
            cout << "\nTitle: " << book.title;
            cout << "\nAuthor: " << book.author;
            cout << "\nGenre: " << book.genre;
            cout << "\nStatus: " << (book.isIssued ? "Issued" : "Available");

            if (book.isIssued) {
                cout << "\nIssued Date: " << ctime(&book.issueDate);
                cout << "Due Date: " << ctime(&book.returnDate);
            }
            cout << endl;
            found = true;
        }
    }

    if (!found) cout << "No books found!\n";
    fin.close();
}
```

If a book is currently issued, the system also displays the issue and due dates. This function makes it easy for users or librarians to locate and track books within the catalog efficiently.

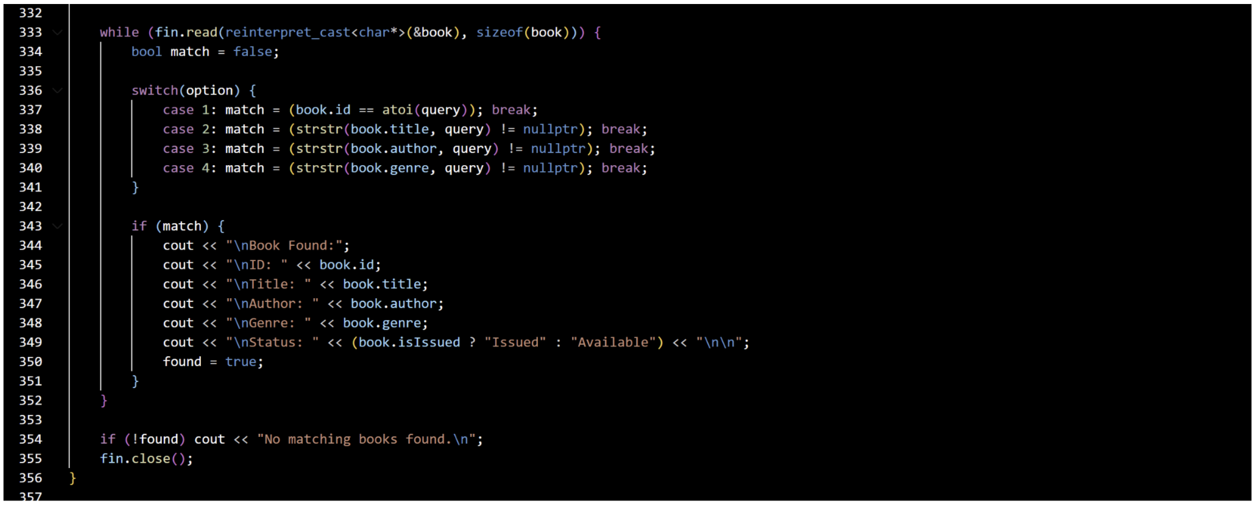
Searching Books

The search function supports flexible lookups by allowing users to search for books by ID, title, author, or genre. It uses partial matching to find relevant results, meaning the user does not need to type the full word or exact phrase.

This enhances usability by mimicking common search functionality found in modern applications and helps users quickly find the information they are looking for.



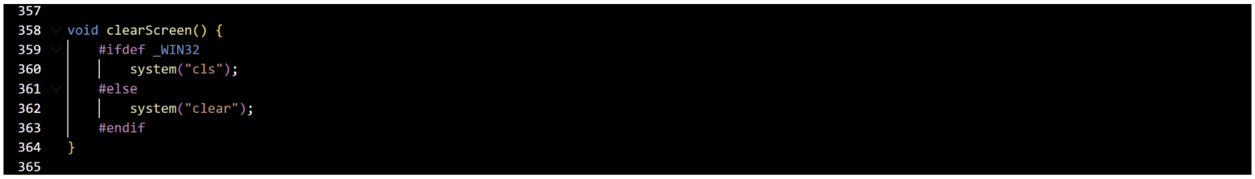
```cpp

void searchBook() {
    int option;
    cout << "\nSearch by:\n1. ID\n2. Title\n3. Author\n4. Genre\n";
    cout << "Enter choice: ";
    inputValidation(option);

    char query[100];
    cout << "Enter search term: ";
    inputValidation(query, 100);

    ifstream fin(FILENAME, ios::binary);
    Book book;
    bool found = false;

```



```cpp

    while (fin.read(reinterpret_cast<char*>(&book), sizeof(book))) {
        bool match = false;

        switch(option) {
            case 1: match = (book.id == atoi(query)); break;
            case 2: match = (strstr(book.title, query) != nullptr); break;
            case 3: match = (strstr(book.author, query) != nullptr); break;
            case 4: match = (strstr(book.genre, query) != nullptr); break;
        }

        if (match) {
            cout << "\nBook Found:";
            cout << "\nID: " << book.id;
            cout << "\nTitle: " << book.title;
            cout << "\nAuthor: " << book.author;
            cout << "\nGenre: " << book.genre;
            cout << "\nStatus: " << (book.isIssued ? "Issued" : "Available") << "\n\n";
            found = true;
        }
    }

    if (!found) cout << "No matching books found.\n";
    fin.close();
}

```

Clearing the Screen

To improve readability, especially during extended sessions, the system includes a utility function to clear the console screen. This function detects the operating system and executes the appropriate command for clearing the screen.



```
void clearScreen() {
    #ifdef _WIN32
        system("cls");
    #else
        system("clear");
    #endif
}
```

Using clearScreen() at the start of each main menu loop or after major operations helps keep the interface clean and focused, especially when working with long lists of output.

Conclusion

This project demonstrates practical applications of several key C++ programming concepts, including binary file handling, modular function design, input validation, structure-based data management, and date/time calculations. By simulating a real-world library management system, it provides students with valuable experience in building structured, user-centered applications. The codebase offers a solid foundation for future enhancements such as graphical interfaces, database connectivity, or multi-user support.

This system is not only a functional academic tool but also a demonstration of how structured logic, thoughtful design, and attention to usability can come together to solve real-world problems using C++.