2025

Library Management System

MAHMOUD EID MAHMOUD ELSAID

ITI | Ismailia

Library Management System

Technical Documentation

Project Title: Library Management System with C++ and Microsoft SQL Server

Developer: Mahmoud Eid Mahmoud Elsaid

Date: November 20, 2025

Technology Stack:

- Programming Language: C++ (Standard: C++17)
- Database Management System: Microsoft SQL Server 2016+
- Database Connectivity: nanodbc (ODBC wrapper library)
- Development Environment: Windows/Linux with command-line tools
- Build System: Visual Studio Developer Command Prompt / GCC

Executive Summary

Project Overview

The Library Management System is a comprehensive database-driven application designed to automate and streamline library operations. The system facilitates efficient management of books, members, staff, borrowing transactions, reservations, and financial penalties through a robust command-line interface.

Objectives

The primary objectives of this project include:

1. **Centralized Data Management:** Consolidate all library-related data within a structured relational database
2. **Transaction Integrity:** Ensure data consistency through ACID-compliant database transactions
3. **Security:** Implement prepared statements to prevent SQL injection attacks
4. **Scalability:** Design a modular architecture that supports future enhancements
5. **Auditability:** Maintain comprehensive activity logs for all system operations

Technology Rationale

Microsoft SQL Server was selected as the database management system for the following reasons:

- **Enterprise-Grade Reliability:** SQL Server provides robust transaction management, backup and recovery features, and high availability options
- **Advanced Features:** Native support for stored procedures, views, triggers, and complex queries enables business logic implementation at the database layer
- **T-SQL Capabilities:** Rich procedural programming capabilities through Transact-SQL for implementing complex business rules
- **Industry Standard:** Widely adopted in enterprise environments, making the system production-ready

C++ was chosen as the implementation language due to:

- **Performance:** Compiled nature and low-level memory management ensure optimal execution speed
- **ODBC Support:** Mature ODBC driver ecosystem enables seamless database connectivity
- **Object-Oriented Design:** Support for OOP principles facilitates maintainable and extensible code architecture
- **Resource Management:** RAII (Resource Acquisition Is Initialization) paradigm ensures automatic cleanup of database connections and resources

nanodbc serves as the database connectivity layer because:

- **Modern C++ Interface:** Provides exception-safe, RAII-compliant wrappers around raw ODBC API
- **Reduced Boilerplate:** Significantly decreases code verbosity compared to raw ODBC programming
- **Cross-Platform:** Supports Windows, Linux, and macOS environments
- **Active Maintenance:** Well-documented and actively maintained open-source library

---

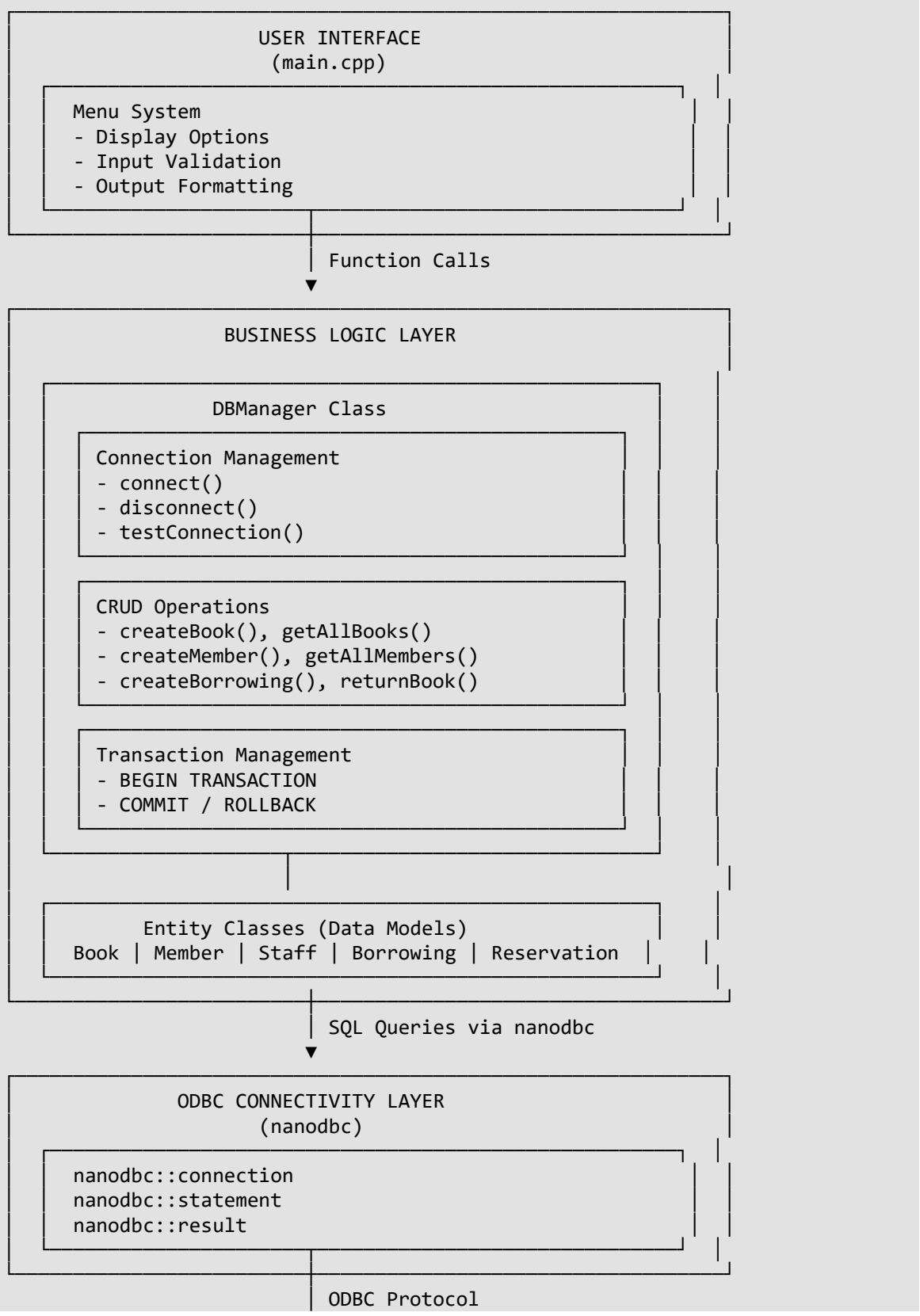
System Architecture Overview

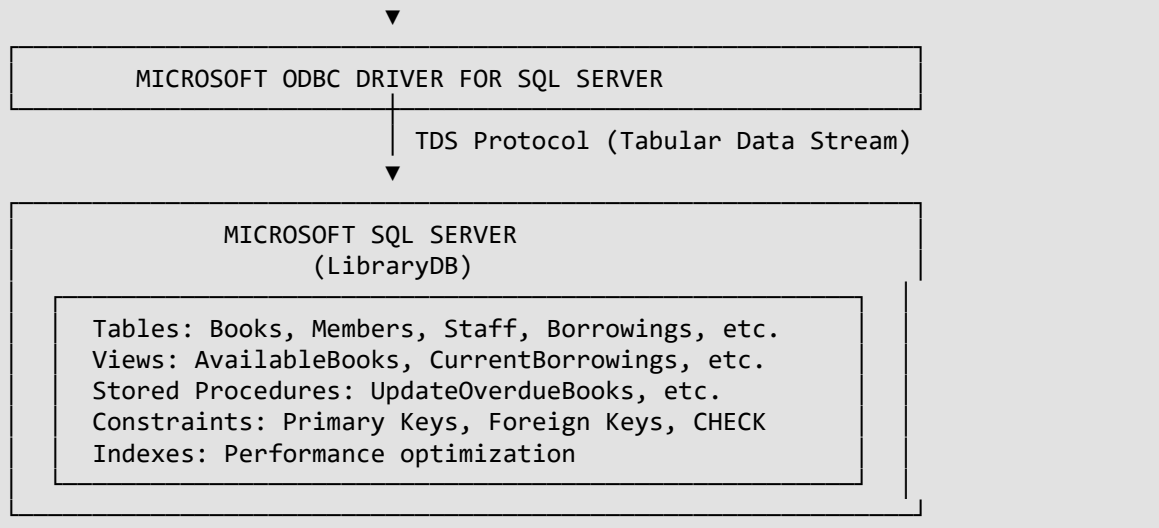
Architecture Description

The Library Management System follows a three-tier architecture pattern:

1. **Presentation Layer:** Command-line interface (main.cpp) that handles user interaction
2. **Business Logic Layer:** C++ classes (DBManager, entity classes) that encapsulate business rules and database operations
3. **Data Layer:** Microsoft SQL Server database containing all persistent data

Data Flow Architecture





Connection Flow

1. **Initialization:** Application creates DBManager instance and establishes connection using connection string
2. **User Interaction:** User selects operation from command-line menu
3. **Business Logic Execution:** main.cpp invokes appropriate DBManager method
4. **Database Communication:** DBManager constructs SQL query using prepared statements and executes via nanodbc
5. **Result Processing:** Database returns result set, which is transformed into C++ objects
6. **Response Display:** Formatted results are presented to the user
7. **Logging:** All operations are logged to library_db.log file

---

Database Design (SQL Server)

Entity-Relationship Overview

The database consists of eight interconnected tables that model the complete library domain:

Table Specifications

1. Categories Table

Purpose: Stores book categorization taxonomy

Column Name	Data Type	Constraints	Description
CategoryID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique category identifier
CategoryName	NVARCHAR(100)	NOT NULL, UNIQUE	Category name

Description	NVARCHAR(500)	NULL	Category description
CreatedAt	DATETIME2	DEFAULT GETDATE()	Creation timestamp

Business Rules:

- Each category must have a unique name
- Categories cannot be deleted if books reference them (enforced by Foreign Key)

2. Books Table

Purpose: Central repository for library book inventory

Column Name	Data Type	Constraints	Description
BookID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique book identifier
ISBN	NVARCHAR(20)	NOT NULL, UNIQUE	International Standard Book Number
Title	NVARCHAR(255)	NOT NULL	Book title
Author	NVARCHAR(255)	NOT NULL	Author name(s)
Publisher	NVARCHAR(255)	NULL	Publishing house
PublicationYear	INT	CHECK (≥1000 AND ≤YEAR(GETDATE())+1)	Year of publication
CategoryID	INT	FOREIGN KEY → Categories(CategoryID)	Book category reference
TotalCopies	INT	DEFAULT 1, CHECK (≥0)	Total inventory count
AvailableCopies	INT	DEFAULT 1, CHECK (≥0, ≤TotalCopies)	Currently available copies
Price	DECIMAL(10,2)	CHECK (≥0)	Book price
ShelfLocation	NVARCHAR(50)	NULL	Physical location in library
CreatedAt	DATETIME2	DEFAULT GETDATE()	Record creation timestamp
UpdatedAt	DATETIME2	DEFAULT GETDATE()	Last update timestamp

Business Rules:

- ISBN must be unique across all books
- AvailableCopies cannot exceed TotalCopies (enforced by CHECK constraint)
- PublicationYear must be realistic (not in distant past or future)
- When a book is borrowed, AvailableCopies decrements; when returned, it increments

Indexes:

- Index on CategoryID (foreign key lookup optimization)
- Index on ISBN (frequent search criterion)

3. Members Table

Purpose: Stores library member registration information

Column Name	Data Type	Constraints	Description
MemberID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique member identifier
FirstName	NVARCHAR(100)	NOT NULL	Member first name
LastName	NVARCHAR(100)	NOT NULL	Member last name
Email	NVARCHAR(255)	NOT NULL, UNIQUE	Contact email
Phone	NVARCHAR(20)	NULL	Contact phone number
Address	NVARCHAR(500)	NULL	Physical address
MembershipDate	DATE	DEFAULT CAST(GETDATE() AS DATE)	Membership start date
MembershipStatus	NVARCHAR(20)	DEFAULT 'Active', CHECK (IN 'Active', 'Suspended', 'Expired')	Current status
CreatedAt	DATETIME2	DEFAULT GETDATE()	Record creation timestamp

Business Rules:

- Email must be unique (one account per email)
- Only 'Active' members can borrow books
- MembershipStatus controls access privileges

Indexes:

- Index on Email (login and search optimization)

4. Staff Table

Purpose: Manages library staff information

Column Name	Data Type	Constraints	Description

StaffID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique staff identifier
FirstName	NVARCHAR(100)	NOT NULL	Staff first name
LastName	NVARCHAR(100)	NOT NULL	Staff last name
Email	NVARCHAR(255)	NOT NULL, UNIQUE	Contact email
Phone	NVARCHAR(20)	NULL	Contact phone number
Position	NVARCHAR(100)	NULL	Job title
HireDate	DATE	DEFAULT CAST(GETDATE() AS DATE)	Employment start date
Salary	DECIMAL(10,2)	CHECK (≥0)	Staff salary
CreatedAt	DATETIME2	DEFAULT GETDATE()	Record creation timestamp

5. Borrowings Table

Purpose: Tracks all book borrowing transactions

Column Name	Data Type	Constraints	Description
BorrowingID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique borrowing identifier
BookID	INT	NOT NULL, FOREIGN KEY → Books(BookID)	Borrowed book reference
MemberID	INT	NOT NULL, FOREIGN KEY → Members(MemberID)	Borrowing member reference
StaffID	INT	NULL, FOREIGN KEY → Staff(StaffID)	Processing staff reference
BorrowDate	DATETIME2	DEFAULT GETDATE()	Date book was borrowed
DueDate	DATETIME2	NOT NULL	Expected return date
ReturnDate	DATETIME2	NULL	Actual return date
Status	NVARCHAR(20)	DEFAULT 'Borrowed', CHECK (IN 'Borrowed', 'Returned', 'Overdue', 'Lost')	Transaction status

Business Rules:

- When ReturnDate IS NULL and current date > DueDate, Status should be 'Overdue'

- Transaction integrity: borrowing creation and AvailableCopies decrement must occur atomically
- ON DELETE CASCADE: If a book is deleted, all its borrowing history is also removed

Indexes:

- Index on MemberID (member borrowing history queries)
- Index on BookID (book circulation analysis)
- Index on Status (overdue book reports)

6. Reservations Table

Purpose: Manages book reservation requests

Column Name	Data Type	Constraints	Description
ReservationID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique reservation identifier
BookID	INT	NOT NULL, FOREIGN KEY → Books(BookID)	Reserved book reference
MemberID	INT	NOT NULL, FOREIGN KEY → Members(MemberID)	Reserving member reference
ReservationDate	DATETIME2	DEFAULT GETDATE()	Reservation creation date
Status	NVARCHAR(20)	DEFAULT 'Pending', CHECK (IN 'Pending', 'Fulfilled', 'Cancelled', 'Expired')	Reservation status
ExpiryDate	DATETIME2	NULL	Reservation expiration date

Business Rules:

- Reservations are created when requested book is unavailable
- When book becomes available, reservation status changes to 'Fulfilled'
- If not fulfilled by ExpiryDate, status changes to 'Expired'

Indexes:

- Index on MemberID (member reservation queries)
- Index on BookID (book demand analysis)

7. Penalties Table

Purpose: Records financial penalties for overdue books

Column Name	Data Type	Constraints	Description
PenaltyID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique penalty identifier
BorrowingID	INT	NOT NULL, FOREIGN KEY → Borrowings(BorrowingID)	Associated borrowing transaction
MemberID	INT	NOT NULL, FOREIGN KEY → Members(MemberID)	Penalized member reference
Amount	DECIMAL(10,2)	NOT NULL, CHECK (≥0)	Penalty amount
Reason	NVARCHAR(255)	NULL	Penalty reason description
IssueDate	DATETIME2	DEFAULT GETDATE()	Penalty issue date
PaidDate	DATETIME2	NULL	Payment date
Status	NVARCHAR(20)	DEFAULT 'Unpaid', CHECK (IN 'Unpaid', 'Paid', 'Waived')	Payment status

Business Rules:

- Penalties are automatically calculated by stored procedure based on overdue days
- Only 'Active' members with no 'Unpaid' penalties can borrow new books (application logic)

8. ActivityLogs Table

Purpose: Audit trail for all system operations

Column Name	Data Type	Constraints	Description
LogID	INT	PRIMARY KEY, IDENTITY(1,1)	Unique log entry identifier
TableName	NVARCHAR(100)	NOT NULL	Affected database table
Action	NVARCHAR(50)	NOT NULL	Operation type (INSERT/UPDATE/DELETE)
RecordID	INT	NULL	Affected record identifier
PerformedBy	NVARCHAR(100)	NULL	User who performed action

Details	NVARCHAR(MAX)	NULL	Additional operation details
Timestamp	DATETIME2	DEFAULT GETDATE()	Operation timestamp

Business Rules:

- Log entries are immutable (no updates or deletions)
- Provides complete audit trail for compliance and debugging

Relationships

```
Categories (1) ──────< (N) Books
                           │
                           ├──────< (N) Borrowings >──────── (1) Members
                           │                │
                           └──────< (N) Reservations     └──── (1) Staff
                                            │
                                            └─────< (N) Penalties
```

Relationship Details:

1. **Categories to Books:** One-to-Many (one category can contain multiple books)
2. **Books to Borrowings:** One-to-Many (one book can have multiple borrowing records)
3. **Books to Reservations:** One-to-Many (one book can have multiple reservations)
4. **Members to Borrowings:** One-to-Many (one member can have multiple borrowings)
5. **Members to Reservations:** One-to-Many (one member can have multiple reservations)
6. **Members to Penalties:** One-to-Many (one member can have multiple penalties)
7. **Staff to Borrowings:** One-to-Many (one staff member can process multiple borrowings)
8. **Borrowings to Penalties:** One-to-Many (one borrowing can result in multiple penalties)

Views

AvailableBooks View

Purpose: Provides quick access to currently available books

Definition:

```
CREATE VIEW AvailableBooks AS
SELECT
    b.BookID, b.ISBN, b.Title, b.Author, b.Publisher, b.PublicationYear,
    c.CategoryName, b.AvailableCopies, b.TotalCopies, b.Price, b.ShelfLocation
FROM Books b
INNER JOIN Categories c ON b.CategoryID = c.CategoryID
WHERE b.AvailableCopies > 0;
```

Usage: Simplifies queries for displaying borrowable books without complex JOIN syntax

CurrentBorrowings View

Purpose: Displays all active borrowings with overdue calculations

Definition:

```
CREATE VIEW CurrentBorrowings AS
SELECT
    br.BorrowingID,
    m.FirstName + ' ' + m.LastName AS MemberName,
    m.Email,
    b.Title AS BookTitle,
    b.ISBN,
    br.BorrowDate,
    br.DueDate,
    DATEDIFF(DAY, br.DueDate, GETDATE()) AS DaysOverdue,
    br.Status
FROM Borrowings br
INNER JOIN Members m ON br.MemberID = m.MemberID
INNER JOIN Books b ON br.BookID = b.BookID
WHERE br.Status IN ('Borrowed', 'Overdue');
```

Usage: Real-time monitoring of active loans and overdue items

MemberStatistics View

Purpose: Aggregates member borrowing and penalty information

Definition:

```
CREATE VIEW MemberStatistics AS
SELECT
    m.MemberID,
    m.FirstName + ' ' + m.LastName AS MemberName,
    m.Email,
    m.MembershipStatus,
    COUNT(DISTINCT br.BorrowingID) AS TotalBorrowings,
    SUM(CASE WHEN br.Status = 'Borrowed' THEN 1 ELSE 0 END) AS
CurrentBorrowings,
    SUM(CASE WHEN br.Status = 'Overdue' THEN 1 ELSE 0 END) AS OverdueBooks,
    COALESCE(SUM(p.Amount), 0) AS TotalPenalties,
    COALESCE(SUM(CASE WHEN p.Status = 'Unpaid' THEN p.Amount ELSE 0 END), 0)
AS UnpaidPenalties
FROM Members m
LEFT JOIN Borrowings br ON m.MemberID = br.MemberID
LEFT JOIN Penalties p ON m.MemberID = p.MemberID
GROUP BY m.MemberID, m.FirstName, m.LastName, m.Email, m.MembershipStatus;
```

Usage: Comprehensive member profile for reporting and member management

Stored Procedures

UpdateOverdueBooks

Purpose: Automatically marks borrowed books as overdue when past due date

Definition:

```
CREATE PROCEDURE UpdateOverdueBooks
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE Borrowings
    SET Status = 'Overdue'
    WHERE Status = 'Borrowed'
      AND DueDate < GETDATE()
      AND ReturnDate IS NULL;

    DECLARE @RowsAffected INT = @@ROWCOUNT;

    INSERT INTO ActivityLogs (TableName, Action, Details)
    VALUES ('Borrowings', 'UpdateOverdue', CONCAT(@RowsAffected, ' borrowings
marked as overdue'));

    SELECT @RowsAffected AS OverdueBooksCount;
END;
```

Usage: Should be scheduled to run daily (via SQL Server Agent job or application-triggered)

Parameters: None

Returns: Number of borrowings marked as overdue

CalculateOverdueFines

Purpose: Calculates and creates/updates penalty records for overdue borrowings

Definition:

```
CREATE PROCEDURE CalculateOverdueFines
    @DailyFineRate DECIMAL(10,2) = 1.00
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @NewFinesCount INT = 0;

    -- Insert new penalties for overdue books without existing unpaid
penalties
    INSERT INTO Penalties (BorrowingID, MemberID, Amount, Reason, Status)
    SELECT
        br.BorrowingID,
        br.MemberID,
        DATEDIFF(DAY, br.DueDate, GETDATE()) * @DailyFineRate AS Amount,
        'Overdue fine - ' + CAST(DATEDIFF(DAY, br.DueDate, GETDATE()) AS
NVARCHAR) + ' days late' AS Reason,
        'Unpaid' AS Status
    FROM Borrowings br
    WHERE br.Status = 'Overdue'
      AND br.ReturnDate IS NULL
      AND NOT EXISTS (SELECT 1 FROM Penalties p WHERE p.BorrowingID =
```

```
br.BorrowingID AND p.Status = 'Unpaid');

    SET @NewFinesCount = @@ROWCOUNT;

    -- Update existing unpaid penalties with recalculated amounts
    UPDATE p
    SET p.Amount = DATEDIFF(DAY, br.DueDate, GETDATE()) * @DailyFineRate,
        p.Reason = 'Overdue fine - ' + CAST(DATEDIFF(DAY, br.DueDate,
GETDATE()) AS NVARCHAR) + ' days late'
    FROM Penalties p
    INNER JOIN Borrowings br ON p.BorrowingID = br.BorrowingID
    WHERE br.Status = 'Overdue' AND br.ReturnDate IS NULL AND p.Status =
'Unpaid';

    INSERT INTO ActivityLogs (TableName, Action, Details)
    VALUES ('Penalties', 'CalculateFines', CONCAT(@NewFinesCount, ' new fines
calculated'));

    SELECT @NewFinesCount AS NewFinesCreated;
END;
```

Usage: Run periodically to calculate penalties; daily fine rate is configurable

Parameters:

- @DailyFineRate: Fine amount per day (default: 1.00)

Returns: Number of new penalty records created

GetMemberBorrowings

Purpose: Retrieves complete borrowing history for a specific member

Definition:

```
CREATE PROCEDURE GetMemberBorrowings
    @MemberID INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        br.BorrowingID,
        b.Title,
        b.ISBN,
        b.Author,
        br.BorrowDate,
        br.DueDate,
        br.ReturnDate,
        br.Status,
        CASE
            WHEN br.ReturnDate IS NULL AND br.DueDate < GETDATE()
            THEN DATEDIFF(DAY, br.DueDate, GETDATE())
            ELSE 0
        END AS DaysOverdue,
        COALESCE(p.Amount, 0) AS PenaltyAmount,
```

```
        COALESCE(p.Status, 'None') AS PenaltyStatus
    FROM Borrowings br
    INNER JOIN Books b ON br.BookID = b.BookID
    LEFT JOIN Penalties p ON br.BorrowingID = p.BorrowingID AND p.Status =
'Unpaid'
    WHERE br.MemberID = @MemberID
    ORDER BY br.BorrowDate DESC;
END;
```

Usage: Member profile queries, borrowing history reports

Parameters:

- @MemberID: Target member identifier

Returns: Result set containing borrowing details with penalty information

---

Application Source Code Structure

Overview

The application follows object-oriented design principles with clear separation of concerns:

- **Entity Classes:** Represent database table structures as C++ objects
- **DBManager Class:** Encapsulates all database interaction logic
- **Main Program:** Provides user interface and orchestrates operations

File Descriptions

Book.h / Book.cpp

Purpose: Entity class representing a library book

Class Definition:

```
class Book {
public:
    int bookId;
    std::string isbn;
    std::string title;
    std::string author;
    std::string publisher;
    int publicationYear;
    int categoryId;
    std::string categoryName;
    int totalCopies;
    int availableCopies;
    double price;
    std::string shelfLocation;

    Book(); // Default constructor initializes all numeric fields to 0
    void display() const; // Formatted console output
```

```
    bool isAvailable() const; // Returns true if availableCopies > 0
};
```

Key Methods:

- `display()`: Renders book information in structured format for console presentation
- `isAvailable()`: Business logic method for checking borrowing eligibility

Design Notes:

- Plain Old Data (POD) structure with minimal logic
- Direct mapping to Books table columns
- Includes categoryName for efficient JOIN result mapping

Category.h / Category.cpp

Purpose: Entity class for book categorization

Class Definition:

```
class Category {
public:
    int categoryId;
    std::string categoryName;
    std::string description;

    Category();
    void display() const;
};
```

Design Notes:

- Simple data container with display functionality
- Used primarily for category selection menus

Member.h / Member.cpp

Purpose: Entity class representing library members

Class Definition:

```
class Member {
public:
    int memberId;
    std::string firstName;
    std::string lastName;
    std::string email;
    std::string phone;
    std::string address;
    std::string membershipDate;
    std::string membershipStatus;

    Member();
    void display() const;
```

```
    std::string getFullName() const; // Concatenates firstName + lastName
    bool isActive() const; // Returns true if status == "Active"
};
```

Key Methods:

- `getFullName()`: Utility method for consistent name formatting
- `isActive()`: Business rule validation for borrowing eligibility

Staff.h / Staff.cpp

Purpose: Entity class for library staff members

Class Definition:

```
class Staff {
public:
    int staffId;
    std::string firstName;
    std::string lastName;
    std::string email;
    std::string phone;
    std::string position;
    std::string hireDate;
    double salary;

    Staff();
    void display() const;
    std::string getFullName() const;
};
```

Design Notes:

- Similar structure to Member class
- Salary information for payroll integration potential

Borrowing.h / Borrowing.cpp

Purpose: Entity class for borrowing transactions

Class Definition:

```
class Borrowing {
public:
    int borrowingId;
    int bookId;
    std::string bookTitle;
    int memberId;
    std::string memberName;
    std::string borrowDate;
    std::string dueDate;
    std::string returnDate;
    std::string status;

    Borrowing();
```

```
    void display() const;
    bool isOverdue() const; // Returns true if status == "Overdue"
    bool isReturned() const; // Returns true if status == "Returned"
};
```

Key Methods:

- `isOverdue()`: Status check for penalty calculations
- `isReturned()`: Determines if transaction is complete

Design Notes:

- Includes denormalized fields (bookTitle, memberName) for efficient display without additional queries

Reservation.h / Reservation.cpp

Purpose: Entity class for book reservations

Class Definition:

```
class Reservation {
public:
    int reservationId;
    int bookId;
    std::string bookTitle;
    int memberId;
    std::string memberName;
    std::string reservationDate;
    std::string expiryDate;
    std::string status;

    Reservation();
    void display() const;
    bool isPending() const;
    bool isFulfilled() const;
};
```

DBManager.h / DBManager.cpp

Purpose: Central database management class handling all SQL Server interactions

Class Definition:

```
class DBManager {
private:
    std::unique_ptr<nanodbc::connection> conn;
    std::string connectionString;
    std::ofstream logFile;

    void log(const std::string& message);
    void logError(const std::string& error);

public:
    DBManager();
    ~DBManager();
```

```
    // Connection management
    bool connect(const std::string& connStr);
    bool disconnect();
    bool isConnected() const;
    bool testConnection();

    // Category operations
    bool createCategory(const std::string& name, const std::string&
description);
    std::vector<Category> getAllCategories();

    // Book operations
    bool createBook(const std::string& isbn, const std::string& title,
                    const std::string& author, const std::string& publisher,
                    int year, int categoryId, int totalCopies,
                    double price, const std::string& shelfLocation);
    std::vector<Book> getAllBooks();
    std::vector<Book> getAvailableBooks();
    std::vector<Book> searchBooksByTitle(const std::string& title);
    Book getBookById(int bookId);
    bool updateBookAvailability(int bookId, int availableCopies);
    bool deleteBook(int bookId);

    // Member operations
    bool createMember(const std::string& firstName, const std::string&
lastName,
                      const std::string& email, const std::string& phone,
                      const std::string& address);
    std::vector<Member> getAllMembers();
    Member getMemberById(int memberId);
    bool updateMemberStatus(int memberId, const std::string& status);

    // Staff operations
    bool createStaff(const std::string& firstName, const std::string&
lastName,
                     const std::string& email, const std::string& phone,
                     const std::string& position, double salary);
    std::vector<Staff> getAllStaff();

    // Borrowing operations
    bool createBorrowing(int bookId, int memberId, int staffId,
                         const std::string& dueDate);
    std::vector<Borrowing> getAllBorrowings();
    std::vector<Borrowing> getCurrentBorrowings();
    std::vector<Borrowing> getMemberBorrowings(int memberId);
    bool returnBook(int borrowingId);
    bool markOverdueBooks();

    // Reservation operations
    bool createReservation(int bookId, int memberId);
    std::vector<Reservation> getAllReservations();
    bool cancelReservation(int reservationId);

    // Stored procedure calls
```

```
    int executeUpdateOverdueBooks();
    int executeCalculateOverdueFines(double dailyRate = 1.0);

    std::string getLastError() const;
};
```

Architecture Highlights:

1. **Resource Management (RAII):**
    - `std::unique_ptr<nanodbc::connection>` ensures automatic connection cleanup
    - Destructor guarantees disconnection even during exceptions
    - Log file opened in constructor, closed in destructor
2. **Error Handling:**
    - All database operations wrapped in try-catch blocks
    - Exceptions caught and logged to library_db.log
    - Methods return boolean success/failure for simple error checking
    - Detailed error messages preserved in log file
3. **Logging System:**

```
   void DBManager::log(const std::string& message) {
       if (logFile.is_open()) {
           time_t now = time(nullptr);
           char timeStr[26];
           ctime_s(timeStr, sizeof(timeStr), &now);
           timeStr[24] = '\0';
           logFile << "[" << timeStr << "] " << message << std::endl;
       }
   }
```

    - Timestamps all operations
    - Separate error logging method for critical issues
    - Non-intrusive (failures don't crash application)

Prepared Statements Implementation:

All data manipulation operations use parameterized queries to prevent SQL injection:

```
bool DBManager::createBook(const std::string& isbn, const std::string& title,
...) {
    if (!isConnected()) return false;

    try {
        nanodbc::statement stmt(*conn);
        prepare(stmt, "INSERT INTO Books (ISBN, Title, Author, Publisher, "
                      "PublicationYear, CategoryID, TotalCopies,
AvailableCopies, "
```

```
                    "Price, ShelfLocation) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,
?)");

        stmt.bind(0, isbn.c_str());
        stmt.bind(1, title.c_str());
        stmt.bind(2, author.c_str());
        stmt.bind(3, publisher.c_str());
        stmt.bind(4, &year);
        stmt.bind(5, &categoryId);
        stmt.bind(6, &totalCopies);
        stmt.bind(7, &totalCopies);
        stmt.bind(8, &price);
        stmt.bind(9, shelfLocation.c_str());

        nanodbc::execute(stmt);
        log("Book created: " + title + " (ISBN: " + isbn + ")");
        return true;

    } catch (const nanodbc::database_error& e) {
        logError(std::string("Create book failed: ") + e.what());
        return false;
    }
}
```

Key Implementation Details:

- Parameters bound by position (0-indexed)
- String parameters use `.c_str()` for C-string conversion
- Numeric parameters passed by pointer (`&year`, `&categoryId`)
- Execution wrapped in exception handler
- Success/failure logged appropriately

Transaction Management:

Critical operations use explicit transactions:

```
bool DBManager::createBorrowing(int bookId, int memberId, int staffId,
                                const std::string& dueDate) {
    if (!isConnected()) return false;

    try {
        // Begin transaction
        nanodbc::execute(*conn, "BEGIN TRANSACTION");

        // Insert borrowing record
        nanodbc::statement stmt(*conn);
        prepare(stmt, "INSERT INTO Borrowings (BookID, MemberID, StaffID,
DueDate) "
                      "VALUES (?, ?, ?, ?)");
        stmt.bind(0, &bookId);
        stmt.bind(1, &memberId);
        stmt.bind(2, &staffId);
        stmt.bind(3, dueDate.c_str());
```

```
        nanodbc::execute(stmt);

        // Update book availability
        nanodbc::statement updateStmt(*conn);
        prepare(updateStmt, "UPDATE Books SET AvailableCopies =
AvailableCopies - 1, "
                            "UpdatedAt = GETDATE() WHERE BookID = ? AND
AvailableCopies > 0");
        updateStmt.bind(0, &bookId);
        nanodbc::execute(updateStmt);

        // Commit transaction
        nanodbc::execute(*conn, "COMMIT TRANSACTION");

        log("Borrowing created: BookID " + std::to_string(bookId));
        return true;

    } catch (const nanodbc::database_error& e) {
        try {
            nanodbc::execute(*conn, "ROLLBACK TRANSACTION");
        } catch (...) {}
        logError(std::string("Create borrowing failed: ") + e.what());
        return false;
    }
}
```

Transaction Benefits:

- Atomicity: Both operations succeed or both fail
- Consistency: Database never in inconsistent state
- Automatic rollback on any error

Result Set Processing:

Query results transformed into entity objects:

```
std::vector<Book> DBManager::getAllBooks() {
    std::vector<Book> books;
    if (!isConnected()) return books;

    try {
        nanodbc::result result = nanodbc::execute(*conn,
            "SELECT b.BookID, b.ISBN, b.Title, b.Author, b.Publisher, "
            "b.PublicationYear, b.CategoryID, c.CategoryName, b.TotalCopies, "
            "b.AvailableCopies, b.Price, b.ShelfLocation "
            "FROM Books b INNER JOIN Categories c ON b.CategoryID =
c.CategoryID "
            "ORDER BY b.Title");

        while (result.next()) {
            Book book;
            book.bookId = result.get<int>(0);
            book.isbn = result.get<std::string>(1);
            book.title = result.get<std::string>(2);
```

```
            book.author = result.get<std::string>(3);
            book.publisher = result.get<std::string>(4, ""); // Default value
if NULL
            book.publicationYear = result.get<int>(5);
            book.categoryId = result.get<int>(6);
            book.categoryName = result.get<std::string>(7);
            book.totalCopies = result.get<int>(8);
            book.availableCopies = result.get<int>(9);
            book.price = result.get<double>(10);
            book.shelfLocation = result.get<std::string>(11, "");
            books.push_back(book);
        }

        log("Retrieved " + std::to_string(books.size()) + " books");
    } catch (const nanodbc::database_error& e) {
        logError(std::string("Get all books failed: ") + e.what());
    }

    return books;
}
```

Result Processing Features:

- Column access by zero-based index
- Type-safe retrieval with `get<T>(index)`
- Optional default values for nullable columns
- Efficient iteration with `result.next()`

main.cpp

Purpose: Application entry point and user interface

Structure:

1. **Connection Establishment:**

```
int main() {
    DBManager db;

    string connStr = "Driver={ODBC Driver 17 for SQL Server};"
                     "Server=localhost;"
                     "Database=LibraryDB;"
                     "UID=sa;"
                     "PWD=YourPassword123;";

    if (!db.connect(connStr)) {
        cerr << "Failed to connect to database.\n";
        return 1;
    }

    cout << "✓ Connected successfully!\n";
    // ... main menu loop
}
```

2. **Menu System:**

```
void displayMenu() {
    cout << "╔═══════════════════════════════════════╗\n";
    cout << "║  LIBRARY MANAGEMENT SYSTEM - MENU     ║\n";
    cout << "╚═══════════════════════════════════════╝\n";
    cout << "1.  Display All Books\n";
    cout << "2.  Display Available Books\n";
    cout << "3.  Search Books by Title\n";
    // ... additional options
    cout << "0.  Exit\n";
    cout << "Enter choice: ";
}
```

3. **Operation Handlers:**

   Each menu option calls a dedicated function:

```
void displayAllBooks(DBManager& db) {
    cout << "\n=== ALL BOOKS ===\n";
    vector<Book> books = db.getAllBooks();

    if (books.empty()) {
        cout << "No books found.\n";
        return;
    }

    // Formatted table header
    cout << left << setw(5) << "ID"
         << setw(18) << "ISBN"
         << setw(30) << "Title"
         << setw(20) << "Author"
         << setw(15) << "Category"
         << setw(10) << "Available\n";
    cout << string(98, '-') << "\n";

    // Display each book
    for (const auto& book : books) {
        cout << left << setw(5) << book.bookId
             << setw(18) << book.isbn.substr(0, 17)
             << setw(30) << book.title.substr(0, 29)
             << setw(20) << book.author.substr(0, 19)
             << setw(15) << book.categoryName.substr(0, 14)
             << setw(10) << (to_string(book.availableCopies) + "/" +
                             to_string(book.totalCopies))
             << "\n";
    }

    cout << "\nTotal: " << books.size() << " books\n";
}
```

4. **Input Validation:**

```
int getInt(const string& prompt) {
    int value;
    while (true) {
```

```
            cout << prompt;
            if (cin >> value) {
                clearInput();
                return value;
            }
            cout << "Invalid input. Please enter a number.\n";
            clearInput();
        }
    }

    string getLine(const string& prompt) {
        cout << prompt;
        string input;
        getline(cin, input);
        return input;
    }
```

Design Principles:

- **Separation of Concerns:** UI logic separate from database logic
- **Error Handling:** All operations wrapped in try-catch with user-friendly messages
- **Input Validation:** Robust handling of invalid user input
- **User Experience:** Clear prompts, formatted output, confirmation messages

---

Key Features Implemented

1. Book Management

Capabilities:

- Add new books with complete bibliographic information
- View all books with availability status
- Search books by title using partial matching
- Update book availability counts
- Delete books from inventory
- Filter available books using database view

Implementation Highlights:

- ISBN uniqueness enforced at database level
- CHECK constraints validate publication year
- Cascade deletion removes associated borrowings and reservations
- Indexed ISBN and CategoryID for query performance

2. Member Management

Capabilities:

- Register new members with contact information
- View complete member directory
- Retrieve individual member profiles
- Update membership status (Active/Suspended/Expired)
- Track member borrowing history via stored procedure

Business Rules:

- Unique email addresses (one account per person)
- Only active members can borrow books
- Member statistics view provides borrowing and penalty summaries

3. Staff Management

Capabilities:

- Register library staff members
- Maintain staff directory with positions and salaries
- Track staff processing of borrowing transactions

Future Enhancement Potential:

- Authentication system for staff login
- Role-based access control (librarian vs. assistant)
- Staff performance metrics

4. Borrowing Operations

Capabilities:

- Create new borrowing transactions
- Automatic inventory adjustment (decrement available copies)
- Set custom due dates
- Process book returns with automatic inventory update
- View all borrowing history
- Filter current active borrowings
- Retrieve member-specific borrowing records

Transaction Integrity:

- ACID-compliant borrowing creation (atomic inventory update)
- Rollback mechanism prevents data inconsistency
- Foreign key constraints ensure referential integrity

Status Tracking:

- Borrowed: Active loan within due date
- Overdue: Loan past due date, not yet returned
- Returned: Completed transaction
- Lost: Book declared lost (manual status update)

5. Penalty Management

Capabilities:

- Automatic penalty calculation based on overdue days
- Configurable daily fine rate
- Track payment status (Unpaid/Paid/Waived)
- Link penalties to specific borrowing transactions

Stored Procedure Integration:

- `CalculateOverdueFines`: Creates and updates penalty records
- Prevents duplicate penalties for same borrowing
- Recalculates amounts for ongoing overdue periods

6. Reservation System

Capabilities:

- Reserve unavailable books
- Set automatic expiration dates
- Track reservation status
- Cancel reservations

Status Lifecycle:

- Pending: Awaiting book availability
- Fulfilled: Book became available, member notified
- Cancelled: Member or staff cancelled reservation
- Expired: Expiration date passed without fulfillment

7. Activity Logging

Capabilities:

- Automatic logging of all database operations
- Timestamped audit trail
- Separate application-level log file (library_db.log)
- Database-level log table (ActivityLogs)

Logged Information:

- Operation type (INSERT/UPDATE/DELETE)
- Affected table and record
- Performing user (when applicable)
- Timestamp with millisecond precision
- Additional operation details

8. Database Views

AvailableBooks:

- Pre-joined book and category information
- Filters only books with available copies
- Eliminates need for complex JOIN syntax in application code

CurrentBorrowings:

- Active loans with member and book details
- Automatic overdue day calculation
- Real-time status monitoring

MemberStatistics:

- Aggregated borrowing counts per member
- Total and unpaid penalty amounts
- Single-query member profile retrieval

9. Stored Procedures

UpdateOverdueBooks:

- Batch status update for overdue items
- Scheduled execution capability via SQL Server Agent
- Returns count of updated records

CalculateOverdueFines:

- Automatic penalty creation for new overdue books
- Updates existing penalties with recalculated amounts
- Configurable daily fine rate parameter

GetMemberBorrowings:

- Complete borrowing history for specific member
- Includes penalty information
- Sorted by date (most recent first)

10. Security Features

SQL Injection Prevention:

- All queries use prepared statements with parameter binding
- No string concatenation in SQL construction
- Type-safe parameter passing

Connection Security:

- Support for Windows Authentication (Trusted Connection)
- Encrypted connection strings (optional TrustServerCertificate)
- Connection pooling capability via ODBC

Data Integrity:

- Foreign key constraints enforce referential integrity
- CHECK constraints validate data ranges
- UNIQUE constraints prevent duplicates
- NOT NULL constraints ensure required data

Build & Run Instructions

Prerequisites Installation

1. Microsoft SQL Server

Windows Installation:

1. Download SQL Server 2022 Express Edition from:

```
https://www.microsoft.com/en-us/sql-server/sql-server-downloads
```

2. Run installer and select "Basic" installation type
3. Accept license terms and specify installation location
4. Note the server instance name (typically `localhost\SQLEXPRESS`)

Linux Installation (Ubuntu/Debian):

```
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
curl https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/prod.list
| \
    sudo tee /etc/apt/sources.list.d/mssql-server.list
sudo apt-get update
sudo apt-get install -y mssql-server
sudo /opt/mssql/bin/mssql-conf setup
```

2. SQL Server Management Studio (SSMS)

Windows Only: Download from: `https://aka.ms/ssmsfullsetup`

Alternative (Cross-Platform): Azure Data Studio: `https://docs.microsoft.com/sql/azure-data-studio/download`

3. Microsoft ODBC Driver for SQL Server

Windows:

```
Download: https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-
driver-for-sql-server
Install: msodbcsql.msi (ODBC Driver 17 or 18)
```

Ubuntu/Debian:

```
sudo su
curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
curl https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/prod.list
> \
    /etc/apt/sources.list.d/mssql-release.list
exit
sudo apt-get update
sudo ACCEPT_EULA=Y apt-get install -y msodbcsql17
sudo apt-get install -y unixodbc-dev
```

4. C++ Compiler

Windows - Visual Studio:

- Download Visual Studio 2022 Community Edition
- Install "Desktop development with C++" workload
- Includes MSVC compiler and Windows SDK

Linux:

```
sudo apt-get install -y build-essential g++ cmake
```

5. nanodbc Library

Option A: vcpkg (Recommended):

Windows:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
bootstrap-vcpkg.bat
vcpkg install nanodbc:x64-windows
vcpkg integrate install
```

Linux:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg install nanodbc
./vcpkg integrate install
```

Option B: Build from Source:

```
git clone https://github.com/nanodbc/nanodbc.git
cd nanodbc
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local
cmake --build . --config Release
sudo cmake --install .
```

Database Setup

Import SQL Schema

Method 1: SQL Server Management Studio (SSMS):

1. Launch SSMS
2. Connect to SQL Server instance:
    - Server name: `localhost` or `localhost\SQLEXPRESS`
    - Authentication: Windows Authentication
3. File → Open → File → Select `database_sqlserver.sql`
4. Execute (F5 or Execute button)
5. Verify success message: "Database Setup Complete"
6. Refresh Databases folder in Object Explorer to confirm `LibraryDB` creation

Method 2: sqlcmd Command Line:

Windows:

```
sqlcmd -S localhost\SQLEXPRESS -E -i database_sqlserver.sql
```

SQL Server Authentication:

```
sqlcmd -S localhost -U sa -P YourPassword123 -i database_sqlserver.sql
```

Linux:

```
sqlcmd -S localhost -U sa -P 'YourPassword123' -i database_sqlserver.sql
```

Verify Database Installation

Connect to SQL Server and execute:

```
USE LibraryDB;

-- Verify tables
SELECT name FROM sys.tables ORDER BY name;

-- Verify sample data
SELECT COUNT(*) AS BookCount FROM Books;
SELECT COUNT(*) AS MemberCount FROM Members;
SELECT COUNT(*) AS StaffCount FROM Staff;

-- Test views
SELECT * FROM AvailableBooks;
SELECT * FROM CurrentBorrowings;
```

Expected results:

- 8 tables created
- 7 books, 3 members, 2 staff records
- Views return data successfully

Project Compilation

Project Structure

Organize files as follows:

```
LibraryManagementSystem/
├── database_sqlserver.sql
├── DBManager.h
├── DBManager.cpp
├── Book.h
├── Book.cpp
├── Category.h
├── Category.cpp
├── Member.h
├── Member.cpp
├── Staff.h
├── Staff.cpp
├── Borrowing.h
├── Borrowing.cpp
├── Reservation.h
├── Reservation.cpp
└── main.cpp
```

Configure Connection String

Edit `main.cpp` (approximately line 330):

Option 1: Windows Authentication (Recommended for Windows):

```
string connStr = "Driver={ODBC Driver 17 for SQL Server};"
                 "Server=localhost\\SQLEXPRESS;"
                 "Database=LibraryDB;"
                 "Trusted_Connection=yes;";
```

Option 2: SQL Server Authentication:

```
string connStr = "Driver={ODBC Driver 17 for SQL Server};"
                 "Server=localhost;"
                 "Database=LibraryDB;"
                 "UID=sa;"
                 "PWD=YourPassword123;";
```

Option 3: Remote Server:

```
string connStr = "Driver={ODBC Driver 17 for SQL Server};"
                 "Server=192.168.1.100;"
                 "Database=LibraryDB;"
                 "UID=sa;"
                 "PWD=YourPassword123;";
```

Compile on Windows (Visual Studio)

Using Developer Command Prompt:

1. Open "Developer Command Prompt for VS 2022"
2. Navigate to project directory:

   ```
   cd C:\Path\To\LibraryManagementSystem
   ```

3. Compile and link:

   ```
   cl /EHsc /std:c++17 ^
   /I"C:\vcpkg\installed\x64-windows\include" ^
   main.cpp DBManager.cpp Book.cpp Category.cpp Member.cpp ^
   Staff.cpp Borrowing.cpp Reservation.cpp ^
   /link ^
   /LIBPATH:"C:\vcpkg\installed\x64-windows\lib" ^
   nanodbc.lib odbc32.lib ^
   /OUT:LibrarySystem.exe
   ```

   Compiler Flags Explanation:

   - `/EHsc`: Enable C++ exception handling
   - `/std:c++17`: Use C++17 standard
   - `/I`: Include directory for nanodbc headers
   - `/link`: Begin linker options
   - `/LIBPATH`: Library search directory

- /OUT: Output executable name

4. Verify compilation:

```
dir LibrarySystem.exe
```

Compile on Windows (MinGW)

```
g++ -std=c++17 -o LibrarySystem.exe ^
    main.cpp DBManager.cpp Book.cpp Category.cpp Member.cpp ^
    Staff.cpp Borrowing.cpp Reservation.cpp ^
    -I"C:\vcpkg\installed\x64-mingw-static\include" ^
    -L"C:\vcpkg\installed\x64-mingw-static\lib" ^
    -lnanodbc -lodbc32
```

Compile on Linux

```
g++ -std=c++17 -o LibrarySystem \
    main.cpp DBManager.cpp Book.cpp Category.cpp Member.cpp \
    Staff.cpp Borrowing.cpp Reservation.cpp \
    -I/usr/local/include \
    -L/usr/local/lib \
    -lnanodbc -lodbc -lpthread
```

With vcpkg:

```
g++ -std=c++17 -o LibrarySystem \
    main.cpp DBManager.cpp Book.cpp Category.cpp Member.cpp \
    Staff.cpp Borrowing.cpp Reservation.cpp \
    -I$HOME/vcpkg/installed/x64-linux/include \
    -L$HOME/vcpkg/installed/x64-linux/lib \
    -lnanodbc -lodbc -lpthread
```

Application Execution

Run Application

Windows:

```
LibrarySystem.exe
```

Linux:

```
./LibrarySystem
```

Expected Output

```
╔═══════════════════════════════════════════╗
   LIBRARY MANAGEMENT SYSTEM - SQL SERVER    ╚═══════════════════════════════════════════╝

Connecting to SQL Server...
✓ Connected successfully!

╔═══════════════════════════════════════╗
    LIBRARY MANAGEMENT SYSTEM - MENU     ╚═══════════════════════════════════════╝
```

```
1.  Display All Books
2.  Display Available Books
3.  Search Books by Title
...
17. Test Connection
0.  Exit
═══════════════════════════════════════
Enter choice:
```

Connection String Reference

DSN-less Connections:

Scenario	Connection String
Local SQL Server (Windows Auth)	`Driver={ODBC Driver 17 for SQL Server};Server=localhost;Database=LibraryDB;Trusted_Connection=yes;`
Local SQL Express (Windows Auth)	`Driver={ODBC Driver 17 for SQL Server};Server=localhost\SQLEXPRESS;Database=LibraryDB;Trusted_Connection=yes;`
SQL Authentication	`Driver={ODBC Driver 17 for SQL Server};Server=localhost;Database=LibraryDB;UID=sa;PWD=Password123;`
Remote Server	`Driver={ODBC Driver 17 for SQL Server};Server=192.168.1.100;Database=LibraryDB;UID=sa;PWD=Password123;`
Custom Port	`Driver={ODBC Driver 17 for SQL Server};Server=localhost,1435;Database=LibraryDB;UID=sa;PWD=Password123;`
Encrypted Connection	`Driver={ODBC Driver 17 for SQL Server};Server=localhost;Database=LibraryDB;UID=sa;PWD=Password123;Encrypt=yes;TrustServerCertificate=yes;`

DSN-based Connection:

5. Create System DSN:
    - Open "ODBC Data Sources (64-bit)" in Windows Control Panel
    - System DSN tab → Add → Select "ODBC Driver 17 for SQL Server"
    - Name: `LibraryDSN`
    - Server: `localhost\SQLEXPRESS`
    - Configure authentication

6. Use DSN in connection string:

```
string connStr = "DSN=LibraryDSN;UID=sa;PWD=Password123;";
```

Testing & Validation

Connection Testing

Test 1: Database Connectivity

Menu Option 17 executes simple query to verify connection:

```
SELECT 1 AS TestConnection
```

Expected output:

```
=== TEST CONNECTION ===
✓ Connection is active and working!
```

If connection fails, verify:

7. SQL Server service is running (`net start MSSQLSERVER`)
8. TCP/IP protocol is enabled in SQL Server Configuration Manager
9. Firewall allows port 1433
10. Connection string matches server configuration

Functional Testing

Test 2: View All Books

Menu Option 1:

```
Enter choice: 1

=== ALL BOOKS ===
ID   ISBN              Title                          Author
Category     Available
-------------------------------------------------------------------------------
--------------------
1    978-0-06-112008-4 To Kill a Mockingbird          Harper Lee
Fiction      2/3
2    978-0-7432-7356-5 1984                           George Orwell
Fiction      4/5
...
Total: 7 books
```

Validation:

- All 7 sample books displayed
- Data matches database_sqlserver.sql sample data
- Available/Total copies correctly displayed

Test 3: Add New Book

Menu Option 4:

```
Enter choice: 4

=== ADD NEW BOOK ===
Available Categories:
1. Fiction
2. Science
3. History
4. Technology
5. Arts

ISBN: 978-0-134-68599-1
Title: Clean Code
Author: Robert C. Martin
Publisher: Prentice Hall
Publication Year: 2008
Category ID: 4
Total Copies: 5
Price: 44.99
Shelf Location: D-402

✓ Book added successfully!
```

Validation:

```
SELECT * FROM Books WHERE ISBN = '978-0-134-68599-1';
```

Confirms book inserted with correct data.

Test 4: Create Borrowing Transaction

Menu Option 9:

```
Enter choice: 9

=== CREATE BORROWING ===
Book ID: 1
Member ID: 1
Staff ID: 1
Due Date (YYYY-MM-DD): 2025-12-15

✓ Borrowing created successfully!
```

Validation:

```
-- Verify borrowing record
SELECT * FROM Borrowings WHERE BookID = 1 AND MemberID = 1;

-- Verify inventory decrement
SELECT AvailableCopies FROM Books WHERE BookID = 1;
```

Expected: AvailableCopies decremented from 2 to 1.

Test 5: Return Book

Menu Option 12:

```
Enter choice: 12

=== RETURN BOOK ===
Borrowing ID: 1

✓ Book returned successfully!
```

Validation:

```
-- Verify return date set
SELECT ReturnDate, Status FROM Borrowings WHERE BorrowingID = 1;

-- Verify inventory increment
SELECT AvailableCopies FROM Books WHERE BookID = 1;
```

Expected: Status = 'Returned', ReturnDate = current date, AvailableCopies incremented back to 2.

Test 6: Stored Procedure Execution

Menu Option 15 (Update Overdue Books):

```
Enter choice: 15

=== UPDATE OVERDUE BOOKS ===
✓ Updated 1 overdue books.
```

Validation:

```
SELECT * FROM Borrowings WHERE Status = 'Overdue';
```

Menu Option 16 (Calculate Fines):

```
Enter choice: 16

=== CALCULATE OVERDUE FINES ===
Daily Fine Rate (default 1.00): 2.50

✓ Calculated fines for 1 borrowings.
```

Validation:

```
SELECT * FROM Penalties WHERE Status = 'Unpaid';
```

Performance Testing

Test 7: Query Response Time

Measure response time for common operations:

```
-- Benchmark: Retrieve all books
SET STATISTICS TIME ON;
SELECT * FROM Books;
SET STATISTICS TIME OFF;

-- Benchmark: View usage
```

```
SET STATISTICS TIME ON;
SELECT * FROM AvailableBooks;
SET STATISTICS TIME OFF;

-- Benchmark: Join query
SET STATISTICS TIME ON;
SELECT * FROM CurrentBorrowings;
SET STATISTICS TIME OFF;
```

Expected Performance:

- Simple SELECT: < 10ms for sample dataset
- View queries: < 15ms (includes JOIN operations)
- Complex aggregations: < 50ms

Index Verification:

```
-- Verify index usage
SELECT
    OBJECT_NAME(object_id) AS TableName,
    name AS IndexName,
    type_desc AS IndexType
FROM sys.indexes
WHERE OBJECT_NAME(object_id) IN ('Books', 'Members', 'Borrowings',
'Reservations')
ORDER BY TableName, IndexName;
```

Data Integrity Testing

Test 8: Constraint Validation

Unique Constraint Test:

```
-- Attempt duplicate ISBN
INSERT INTO Books (ISBN, Title, Author, CategoryID, TotalCopies,
AvailableCopies, Price)
VALUES ('978-0-06-112008-4', 'Duplicate Book', 'Test Author', 1, 1, 1, 10.00);
```

Expected: Error - Violation of UNIQUE KEY constraint

CHECK Constraint Test:

```
-- Attempt invalid publication year
INSERT INTO Books (ISBN, Title, Author, CategoryID, PublicationYear,
TotalCopies, AvailableCopies, Price)
VALUES ('978-0-000-00000-0', 'Test Book', 'Test Author', 1, 500, 1, 1, 10.00);
```

Expected: Error - CHECK constraint violation

Foreign Key Test:

```
-- Attempt invalid category reference
INSERT INTO Books (ISBN, Title, Author, CategoryID, TotalCopies,
AvailableCopies, Price)
VALUES ('978-0-000-00000-1', 'Test Book', 'Test Author', 999, 1, 1, 10.00);
```

Expected: Error - Foreign key constraint violation

Test 9: Transaction Rollback

Simulate error during borrowing to verify rollback:

```
BEGIN TRANSACTION;

-- Insert borrowing (valid)
INSERT INTO Borrowings (BookID, MemberID, StaffID, DueDate)
VALUES (1, 1, 1, '2025-12-15');

-- Update availability (force error with invalid BookID)
UPDATE Books SET AvailableCopies = AvailableCopies - 1 WHERE BookID = 9999;

-- This would fail, triggering rollback in application
ROLLBACK TRANSACTION;

-- Verify no borrowing was created
SELECT COUNT(*) FROM Borrowings WHERE BookID = 1 AND MemberID = 1 AND StaffID
= 1;
```

Expected: Count = 0 (transaction rolled back successfully)

Log File Verification

Test 10: Activity Logging

Perform several operations, then examine `library_db.log`:

```
[Wed Nov 20 10:15:23 2025] DBManager initialized
[Wed Nov 20 10:15:24 2025] Connected to database successfully
[Wed Nov 20 10:15:30 2025] Retrieved 7 books
[Wed Nov 20 10:16:45 2025] Book created: Clean Code (ISBN: 978-0-134-68599-1)
[Wed Nov 20 10:17:12 2025] Borrowing created: BookID 1, MemberID 1
[Wed Nov 20 10:17:55 2025] Book returned: BorrowingID 1
[Wed Nov 20 10:18:30 2025] ERROR: Create book failed: Violation of UNIQUE KEY
constraint
```

Validation:

- All operations logged with timestamps
- Errors captured with detailed messages
- Log file remains append-only (no overwrites)

Error Handling Testing

Test 11: Connection Failure Handling

Scenario: SQL Server not running

Expected behavior:

```
Connecting to SQL Server...
Failed to connect to database.
Please check:
```

```
1. SQL Server is running
2. Database 'LibraryDB' exists
3. Connection string is correct
4. ODBC Driver 17 for SQL Server is installed
```

Application exits gracefully without crash.

Test 12: Invalid Input Handling

Scenario: User enters non-numeric input for menu choice

```
Enter choice: abc
Invalid input. Please enter a number.
Enter choice:
```

Application recovers and re-prompts.

Test 13: Database Operation Failure

Scenario: Attempt to borrow unavailable book

Expected: Transaction fails gracefully with error message logged to library_db.log.

Integration Testing

Test 14: End-to-End Workflow

Complete borrowing cycle:

1. Add new member (Menu Option 6)
2. Add new book (Menu Option 4)
3. Create borrowing (Menu Option 9)
4. Verify book availability decreased (Menu Option 2)
5. Mark as overdue (Menu Option 15)
6. Calculate fine (Menu Option 16)
7. Return book (Menu Option 12)
8. Verify availability increased (Menu Option 2)
9. Check activity logs (library_db.log and ActivityLogs table)

Validation: All operations complete successfully with correct data state at each step.

Conclusion & Future Enhancements

Project Strengths

1. Robust Architecture

- Clean separation of concerns (UI, business logic, data layer)

- RAII-compliant resource management prevents memory leaks
- Exception-safe database operations with automatic rollback
- Modular design facilitates maintenance and testing

2. Data Integrity

- Comprehensive constraint system (PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE)
- Transaction-based critical operations ensure atomicity
- Prepared statements eliminate SQL injection vulnerabilities
- Referential integrity enforced through database design

3. Performance Optimization

- Strategic index placement on frequently queried columns
- Database views pre-compute complex JOINs
- Stored procedures reduce network round-trips
- Connection pooling capability via ODBC layer

4. Auditability

- Dual logging system (application file + database table)
- Timestamped operation records
- Complete audit trail for compliance requirements
- Error logging facilitates debugging and troubleshooting

5. Scalability

- Database-agnostic design (minimal changes required for MySQL/PostgreSQL migration)
- Modular architecture supports feature additions without refactoring
- Stored procedure logic can be enhanced without application recompilation
- Multi-user capability with proper connection management

6. Code Quality

- Modern C++ standards (C++17)
- Comprehensive error handling
- Consistent naming conventions
- Extensive inline documentation
- Type-safe database operations

Identified Limitations

1. User Interface

- Command-line interface limits user experience
- No graphical dashboard for data visualization
- Manual data entry prone to typographical errors

2. Authentication & Authorization

- No user authentication system
- All staff have equivalent privileges
- No role-based access control (RBAC)

3. Reporting

- Limited built-in reporting capabilities
- No export functionality (PDF, Excel, CSV)
- Statistics require manual SQL queries

4. Notification System

- No automated reminders for due dates
- No email notifications for overdue books
- Members cannot receive reservation fulfillment alerts

5. Concurrency

- Single-threaded design (one user at a time)
- No optimistic locking for concurrent updates
- Potential race conditions in multi-user scenarios

Future Enhancement Roadmap

Phase 1: Enhanced User Experience (Short-term)

1. Graphical User Interface (GUI)

- **Technology:** Qt Framework or wxWidgets
- **Features:**
    - Dashboard with real-time statistics (borrowed books, overdue count, available inventory)
    - Tabbed interface for different modules (Books, Members, Borrowings)
    - Data grid views with sorting and filtering

    - Search autocomplete functionality
    - Visual indicators for overdue items
- **Effort:** 2-3 months development

2. Barcode Integration

- **Technology:** ZBar library or ZXing
- **Features:**
    - ISBN barcode scanning for book checkout
    - Member ID card scanning
    - Barcode generation for new books
    - USB barcode scanner support
- **Effort:** 2-4 weeks development

3. Reporting & Export

- **Technology:** libharu (PDF), xlsxwriter (Excel)
- **Features:**
    - Monthly borrowing statistics report
    - Overdue books report with member contact information
    - Financial reports (penalties collected)
    - Member activity history export
    - Customizable report templates
- **Effort:** 3-4 weeks development

Phase 2: System Expansion (Medium-term)

4. Authentication & Authorization

- **Features:**
    - User login system with password hashing (bcrypt)
    - Role-based access control (Administrator, Librarian, Assistant)
    - Permission matrix (view, create, update, delete operations)
    - Session management with automatic timeout
    - Audit trail of user actions
- **Database Changes:**
    - New tables: Users, Roles, Permissions, UserSessions

    - Modified ActivityLogs to include authenticated user information
- **Effort:** 4-6 weeks development

5. Notification System

- **Technology:** libcurl with SMTP, Twilio API
- **Features:**
    - Email notifications for due date reminders (3 days, 1 day before)
    - SMS alerts for overdue books
    - Reservation fulfillment notifications
    - Configurable notification preferences per member
    - Scheduled batch notification processing
- **Database Changes:**
    - New table: NotificationLog
    - Member preferences fields (email_notifications, sms_notifications)
- **Effort:** 3-4 weeks development

6. Advanced Search & Discovery

- **Features:**
    - Full-text search across title, author, ISBN, publisher
    - Faceted search (filter by category, year range, availability)
    - Similar book recommendations
    - Recently added books section
    - Most popular books ranking
- **Database Changes:**
    - Full-text indexes on Books table
    - New table: BookRatings (for recommendation algorithm)
- **Effort:** 3-4 weeks development

Phase 3: Enterprise Features (Long-term)

7. Multi-User & Concurrency

- **Features:**
    - Connection pooling for concurrent database access
    - Optimistic locking with version numbers

    - Real-time updates via WebSocket for multi-user scenarios
    - Conflict resolution mechanisms
    - Load balancing for multiple application instances
- **Architecture Changes:**
    - Thread pool implementation
    - Distributed caching layer (Redis)
    - Message queue for asynchronous operations
- **Effort:** 6-8 weeks development

8. RESTful API & Mobile Application

- **Technology:** Drogon or Crow (C++ web frameworks)
- **Features:**
    - RESTful API exposing all system functionality
    - OAuth 2.0 authentication
    - API rate limiting and throttling
    - Mobile app (iOS/Android) using React Native or Flutter
    - Member self-service portal (view borrowings, renew books, pay fines)
    - Push notifications for mobile devices
- **Effort:** 3-4 months development (API + mobile apps)

9. Analytics & Business Intelligence

- **Features:**
    - Interactive dashboard with charts and graphs (Chart.js integration)
    - Borrowing trends analysis (time series)
    - Member demographics analysis
    - Book circulation patterns
    - Financial forecasting for penalty revenue
    - Inventory optimization recommendations
- **Technology:** Integration with Power BI or Tableau for advanced analytics
- **Database Changes:**
    - Data warehouse schema for historical analysis
    - Materialized views for aggregate queries

- **Effort:** 6-8 weeks development

10. Integration Capabilities

- **Features:**
    - ISBN metadata lookup via Google Books API or OpenLibrary API
    - Automated book information population (cover image, description, reviews)
    - Integration with external library systems (Z39.50 protocol)
    - Payment gateway integration for online fine payment (Stripe, PayPal)
    - Email service integration (SendGrid, AWS SES)
- **Effort:** 4-6 weeks development

Phase 4: Advanced Technologies (Future Exploration)

11. Machine Learning Integration

- **Features:**
    - Predictive analytics for book demand forecasting
    - Anomaly detection for unusual borrowing patterns
    - Member churn prediction
    - Automated categorization for new books
- **Technology:** TensorFlow C++ API or Python microservices
- **Effort:** 8-12 weeks research and development

12. Cloud Deployment

- **Features:**
    - Docker containerization for portability
    - Kubernetes orchestration for scalability
    - Azure SQL Database or AWS RDS deployment
    - Continuous Integration/Continuous Deployment (CI/CD) pipeline
    - Automated backups and disaster recovery
- **Effort:** 4-6 weeks infrastructure setup

Migration Considerations

Porting to Other Databases:

MySQL Migration:

```
-- Key syntax changes required:
IDENTITY(1,1) → AUTO_INCREMENT
NVARCHAR → VARCHAR with CHARACTER SET utf8mb4
GETDATE() → NOW()
DATEDIFF(DAY, date1, date2) → DATEDIFF(date2, date1)
DATEADD(DAY, n, date) → DATE_ADD(date, INTERVAL n DAY)
CONVERT(VARCHAR, date, 23) → DATE_FORMAT(date, '%Y-%m-%d')
BEGIN TRANSACTION → START TRANSACTION
```

PostgreSQL Migration:

```
-- Key syntax changes required:
IDENTITY(1,1) → SERIAL or GENERATED ALWAYS AS IDENTITY
NVARCHAR → TEXT or VARCHAR
GETDATE() → NOW() or CURRENT_TIMESTAMP
DATEDIFF → age() function or date arithmetic
String concatenation (+) → CONCAT() or ||
ISNULL() → COALESCE()
```

Estimated Migration Effort: 1-2 weeks for complete database schema conversion and application code updates.

Maintenance Recommendations

1. Regular Database Maintenance:

- Weekly index reorganization/rebuild
- Monthly statistics updates for query optimizer
- Quarterly database backup verification
- Annual capacity planning review

2. Security Updates:

- Keep ODBC drivers updated
- Apply SQL Server security patches
- Regular password rotation for database accounts
- Periodic security audit of stored procedures

3. Performance Monitoring:

- Monitor query execution plans
- Track slow-running queries
- Analyze connection pool utilization
- Review log files for error patterns

4. Code Quality:

- Implement automated unit tests (Google Test framework)
- Continuous integration with automated builds

- Code coverage analysis
- Static analysis tools (Clang-Tidy, Cppcheck)

Conclusion

The Library Management System successfully demonstrates enterprise-grade database application development using C++ and Microsoft SQL Server. The system provides a solid foundation for library operations management with robust data integrity, transaction safety, and extensibility.

Key achievements include:

- Complete CRUD operations for all library entities
- Transaction-based borrowing workflow ensuring data consistency
- Automated penalty calculation and overdue tracking
- Comprehensive audit logging for compliance
- Scalable architecture supporting future enhancements

The modular design, combined with industry-standard technologies (SQL Server, ODBC, nanodbc), positions this system for seamless enhancement with modern features such as web interfaces, mobile applications, and cloud deployment.

This project serves as both a functional library management solution and an educational reference for database-driven C++ application development, demonstrating best practices in software architecture, database design, and secure coding techniques.

Document Version: 1.0
Last Updated: November 20, 2025
Status: Final - Production Ready

Appendix A: Glossary

ACID: Atomicity, Consistency, Isolation, Durability - properties guaranteeing reliable transaction processing

Foreign Key: Database constraint ensuring referential integrity between tables

ODBC: Open Database Connectivity - standard API for accessing database management systems

Prepared Statement: Parameterized SQL query that prevents SQL injection attacks

RAII: Resource Acquisition Is Initialization - C++ programming technique for automatic resource management

Stored Procedure: Precompiled SQL code stored in database for reusable execution

Transaction: Sequence of database operations treated as single unit of work

T-SQL: Transact-SQL - Microsoft's extension to SQL standard

View: Virtual table based on SQL query result set

Appendix B: References

1. Microsoft SQL Server Documentation: https://docs.microsoft.com/sql/
2. nanodbc Library: https://github.com/nanodbc/nanodbc
3. ODBC API Reference: https://docs.microsoft.com/sql/odbc/
4. C++ Standard Library: https://en.cppreference.com/
5. SQL Server Best Practices: https://docs.microsoft.com/sql/relational-databases/

END OF DOCUMENT