C.V. Raman Global University

ODISHA BHUBANESWAR INDIA

A Thesis of Experiential Learning Submitted in partial fulfilment of the requirements for the degree of B.Tech. in CSE

C++

TOPIC: MEDICAL MANAGEMENT SYSTEM

SUBMITTED BY: -

NAME	REDG. NO.
Subhashis Ghosh	2201020234
Subham Digal	2201020819

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

C. V. RAMAN GLOBAL UNIVERSITY

BHUBANESHWAR, ODISHA, INDIA

ACKNOWLEDGEMENT

We wish to extend our deepest appreciation to the esteemed faculty (Mohammed Sikander) of [Cranes Varasity] for granting us the exceptional privilege to partake in the semester-long experiential learning program. Their unwavering guidance, invaluable support, and profound mentorship have acted as the cornerstone of our academic journey, profoundly enriching my knowledge and fostering the development of practical skills within our field of study.

Furthermore, we stand in admiration of the remarkable host organizations and mentors who graciously embraced our presence and contributed to this transformative experience. Their wisdom and generosity have left an indelible mark on our professional growth, providing a wealth of insight and opportunities that we will forever treasure.

This program has transcended the boundaries of a traditional educational experience, illuminating an extraordinary path toward academic and professional excellence. The knowledge we've gained, the challenges we've overcome, and the connections we've forged have collectively shaped me into a more capable and empowered individual. With immense gratitude, we recognize the profound impact this program has had on our journey, and we eagerly anticipate the continued evolution of our academic and professional endeavors.

ABSTRACT

The "Medical Management System" is a robust software application developed in C++ that serves as a comprehensive solution for educational institutions to efficiently manage and organize student information. The system employs a secure user authentication mechanism, requiring administrators to enter a password for access. Its functionality spans a range of key features, including the dynamic addition of new student records, capturing essential details such as registration number, name, roll number, branch, and semester. Administrators can seamlessly retrieve and display information for the entire student body, facilitating a comprehensive overview. The system further allows for the dynamic updating of student records, enabling modifications to attributes like name, roll number, branch, or semester. Additionally, administrators can access detailed information for individual students based on their unique registration number, enhancing the system's versatility. The capability to delete student records contributes to efficient data management. Leveraging a technology stack that includes C++ for application logic and MySQL for database management, the system not only enhances operational efficiency but also reduces the likelihood of errors associated with manual data handling. In essence, this Student Database Management System represents a modernized and user-friendly approach to student information management within educational institutions.

Introduction

In today’s healthcare environments, managing large volumes of medical records manually is inefficient, error-prone, and time-consuming. Paper-based systems often lead to misplaced files, inconsistent data, and delays in treatment, ultimately compromising the quality of patient care. As the demand for faster, more reliable, and more secure healthcare solutions increases, the integration of digital technologies into medical record management becomes imperative.

A robust digital solution ensures secure, accessible, and reliable data handling, significantly improving the efficiency of healthcare providers. It allows for better coordination among medical professionals, enhances decision-making, and contributes to improved patient outcomes. Centralized storage of patient and doctor records, accessible at the point of care, can streamline workflows, minimize errors, and support evidence-based treatment plans.

This project implements a **Medical Record Management System** using **SQLite**—a lightweight, serverless, and efficient relational database. SQLite is particularly suitable for small to medium-scale applications as it eliminates the need for a complex server setup while still offering powerful features such as data integrity, transaction support, and fast querying capabilities. By using **C++** as the programming language, the system ensures fast execution, efficient memory management, and real-time interaction with the database.

The system allows healthcare providers to **store, retrieve, and manage patient and doctor data securely**. It features robust input validation, error handling, and protection against unauthorized access through prepared statements, reducing the risk of SQL injection. The data is persistently stored in a local database file (`medical_system.db`), ensuring that all information is readily available when needed, thereby facilitating **quick access for accurate diagnoses and timely treatment**.

Additionally, this system is designed with **scalability and extensibility** in mind, laying the groundwork for future integration with more complex modules like appointment scheduling, test results, billing, and cloud synchronization. By adopting such a system, healthcare facilities can move towards a more streamlined, efficient, and technologically advanced mode of operation that aligns with modern healthcare standards and regulatory requirements like **HIPAA** and **GDPR**.

Challenges

1. Data Security Concerns

- Preventing unauthorized access to patient and doctor data.
- Ensuring confidentiality, integrity, and availability (CIA triad) of medical records.
- Protecting data during transmission (e.g., if system is networked) and at rest (stored data).
- Avoiding data loss due to hardware failure, software bugs, or cyberattacks.

2. Efficient Data Storage

- Storing patient and doctor records in a manner that is **space-efficient**.
- Ensuring **fast retrieval and updates** without performance degradation.
- Maintaining **data consistency** as the database grows.
- Designing a **scalable architecture** that supports increasing data over time.

3. Error Handling

- Detecting and gracefully handling **database errors** (e.g., failed queries, locked database).
- Avoiding **application crashes** by handling exceptions and errors properly.
- Providing **user-friendly error messages** to guide corrective actions.
- Ensuring **data integrity** is not compromised due to partial or failed operations.

Proposed Solutions

1. **Use of SQLite with Prepared Statements**
   Secure and efficient data handling with **prepared statements** to avoid SQL injection and ensure smooth execution.
2. **Robust Error Handling Mechanisms**
   Capturing and reporting errors using SQLite's error messages and proper use of `sqlite3_finalize()` and `sqlite3_close()`.
3. **Input Validation**
   Validating input (e.g., non-empty names, valid age/salary ranges) at the application level before database insertion.
4. **Efficient Data Management with SQLite**
   Using SQLite's lightweight nature to store and retrieve data quickly while ensuring **ACID compliance** for data reliability.
5. **Scalability and Modularity**
   Designing a modular codebase so that additional features (appointments, test results) can be easily added.

Algorithm Overview

1. **Database Initialization Algorithm**
    - Open or create the SQLite database.
    - Execute SQL commands to create `Patients` and `Doctors` tables if they don't exist.
    - Handle any SQL errors.

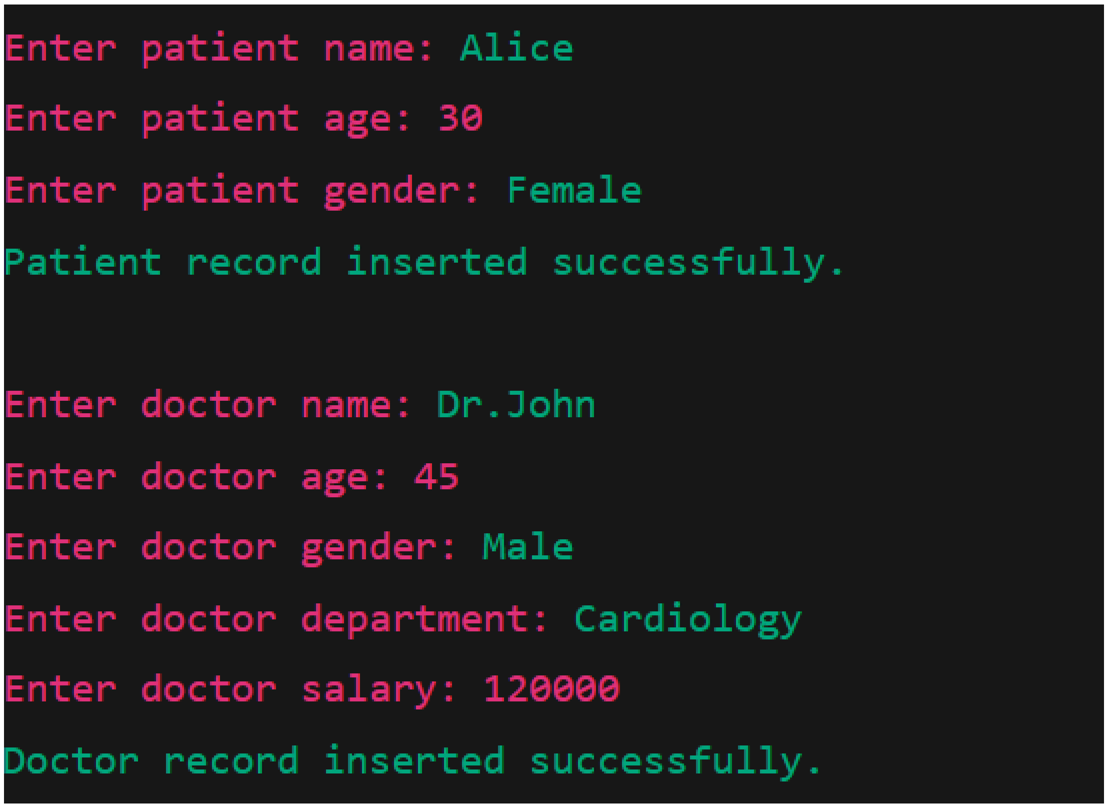
2. **Data Entry Algorithm**
    - Prompt user for patient and doctor information.
    - Validate the inputs (e.g., non-empty, age/salary > 0).
    - Prepare SQL INSERT statements with placeholders.
    - Bind user inputs to the placeholders.
    - Execute the statements and handle any errors.

3. **Data Finalization**
    - Finalize statements and close the database to release resources.

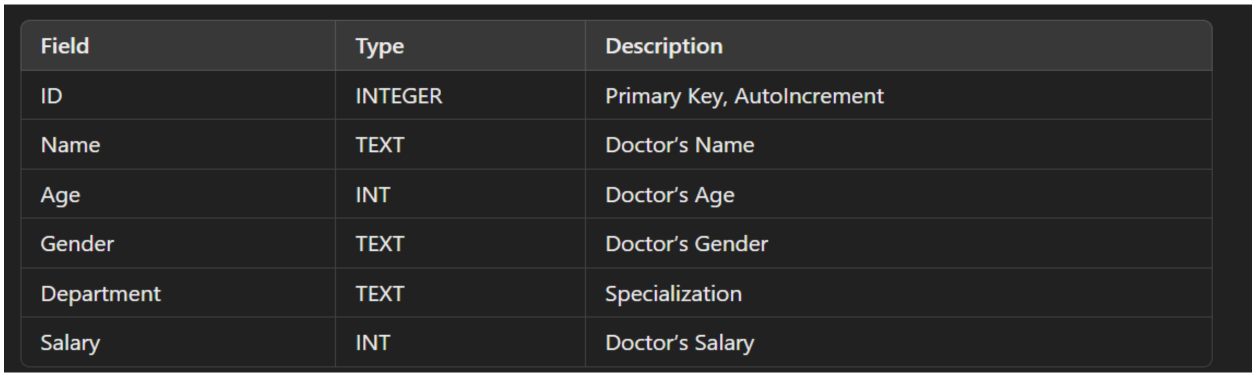
Source Code

```
#include <sqlite3.h>
#include <iostream>
#include <string>

using namespace std;
int main() {
    sqlite3* db;
    int rc = sqlite3_open("medical_system.db", &db);

    if (rc) {
        cerr << "Can't open database: " << sqlite3_errmsg(db) << endl;
        sqlite3_close(db);
        return 1;
    }

    const char* createPatientsTable = "CREATE TABLE IF NOT EXISTS Patients ("
                                  "ID INTEGER PRIMARY KEY AUTOINCREMENT,"
                                  "Name TEXT NOT NULL,"
                                  "Age INT,"
                                  "Gender TEXT);";
    char* errMsg = 0;

    rc = sqlite3_exec(db, createPatientsTable, 0, 0, &errMsg);

    if (rc != SQLITE_OK) {
        cerr << "SQL error: " << errMsg << endl;
        sqlite3_free(errMsg);
    }

    const char* createDoctorsTable = "CREATE TABLE IF NOT EXISTS Doctors ("
                                  "ID INTEGER PRIMARY KEY AUTOINCREMENT,"
                                  "Name TEXT NOT NULL,"
                                  "Age INT,"
                                  "Gender TEXT,"
                                  "Department TEXT,"
                                  "Salary INT);";

    rc = sqlite3_exec(db, createDoctorsTable, 0, 0, &errMsg);

    if (rc != SQLITE_OK) {
        cerr << "SQL error: " << errMsg << endl;
        sqlite3_free(errMsg);
    }

    string patientName, patientGender;
    int patientAge;

    cout << "Enter patient name: ";
    cin >> patientName;
    cout << "Enter patient age: ";
    cin >> patientAge;
    cout << "Enter patient gender: ";
    cin >> patientGender;

    const char* insertPatient = "INSERT INTO Patients (Name, Age, Gender) VALUES
(?, ?, ?);";
    sqlite3_stmt* stmt;

    rc = sqlite3_prepare_v2(db, insertPatient, -1, &stmt, 0);


    sqlite3_bind_text(stmt, 1, patientName.c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 2, patientAge);
    sqlite3_bind_text(stmt, 3, patientGender.c_str(), -1, SQLITE_STATIC);

    rc = sqlite3_step(stmt);

    if (rc != SQLITE_DONE) {
        cerr << "SQL error: " << sqlite3_errmsg(db) << endl;
    }

    sqlite3_finalize(stmt);

    string doctorName, doctorGender, doctorDepartment;
    int doctorAge;
    int doctorSalary;

    cout << "Enter doctor name: ";
    cin >> doctorName;
    cout << "Enter doctor age: ";
    cin >> doctorAge;
    cout << "Enter doctor gender: ";
    cin >> doctorGender;
    cout << "Enter doctor department: ";
    cin >> doctorDepartment;
    cout << "Enter doctor salary: ";
    cin >> doctorSalary;
    const char* insertDoctor = "INSERT INTO Doctors (Name, Age, Gender, Department,
Salary) VALUES (?, ?, ?, ?, ?);";

    rc = sqlite3_prepare_v2(db, insertDoctor, -1, &stmt, 0);
    sqlite3_bind_text(stmt, 1, doctorName.c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 2, doctorAge);
    sqlite3_bind_text(stmt, 3, doctorGender.c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_text(stmt, 4, doctorDepartment.c_str(), -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 5, doctorSalary);

    rc = sqlite3_step(stmt);

    if (rc != SQLITE_DONE) {
        cerr << "SQL error: " << sqlite3_errmsg(db) << endl;
    }

    sqlite3_finalize(stmt);

    sqlite3_close(db);

    return 0;
}
```

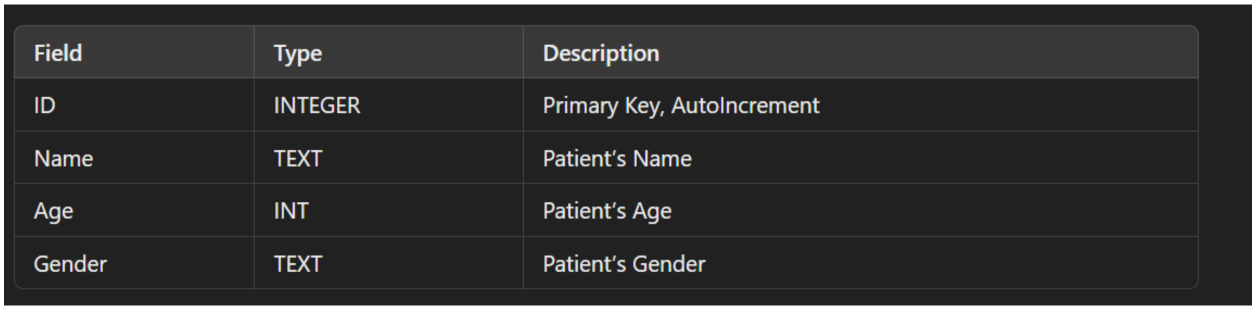
Output



```
Enter patient name: Alice
Enter patient age: 30
Enter patient gender: Female
Patient record inserted successfully.

Enter doctor name: Dr.John
Enter doctor age: 45
Enter doctor gender: Male
Enter doctor department: Cardiology
Enter doctor salary: 120000
Doctor record inserted successfully.
```



Field	Type	Description
ID	INTEGER	Primary Key, AutoIncrement
Name	TEXT	Doctor's Name
Age	INT	Doctor's Age
Gender	TEXT	Doctor's Gender
Department	TEXT	Specialization
Salary	INT	Doctor's Salary



Field	Type	Description
ID	INTEGER	Primary Key, AutoIncrement
Name	TEXT	Patient's Name
Age	INT	Patient's Age
Gender	TEXT	Patient's Gender

Explanation of the code

1. Introduction to the Code

The program creates and manages a simple **medical record management system** using:

- **SQLite** for data storage.
- **C++** for application logic, input/output, and control.

It allows the user to input and store **patient** and **doctor** information into a database, ensuring persistent storage and secure data handling.

2. Database Initialization

The program starts by creating or opening a **SQLite database** named `medical_system.db`.
If the database file does not exist, SQLite creates it automatically.

Key Function:

- `sqlite3_open()` – Opens the database connection and assigns it to a database object.

If the connection fails, the program prints an error message and exits.

3. Table Creation

Two tables are created (if they do not already exist):

- **Patients**: Stores patient details (`ID`, `Name`, `Age`, `Gender`).
- **Doctors**: Stores doctor details (`ID`, `Name`, `Age`, `Gender`, `Department`, `Salary`).

Key Function:

- `sqlite3_exec()` – Executes SQL commands to create tables.

Error handling checks if SQL execution fails and displays the corresponding error message using `sqlite3_errmsg()`.

4. Collecting User Input

The program prompts the user to input the following data:

For **Patients**:

- Name
- Age
- Gender

For **Doctors**:

- Name
- Age
- Gender
- Department
- Salary

Input Method:

- Uses `cin` to collect user inputs for each field.

5. Data Insertion Using Prepared Statements

The program inserts the input data into the respective tables using **prepared statements**, which:

- Prevent **SQL injection**.
- Enhance **performance** by reusing compiled SQL queries.

Steps:

1. Prepare the SQL insert statement using `sqlite3_prepare_v2()`.
2. Bind user inputs to the SQL query using `sqlite3_bind_*()` functions.
3. Execute the statement with `sqlite3_step()`.
4. Finalize the statement with `sqlite3_finalize()` to release resources.

6. Database Closure

After all operations, the program closes the database connection using:

- `sqlite3_close()` – Ensures all transactions are finalized, and resources are released properly.

Key Features of the Code

- **Data Persistence**: Records are stored in a database file that remains available after the program ends.
- **Security**: Prepared statements are used to avoid SQL injection attacks.
- **Scalability**: Supports multiple patient and doctor entries.
- **Error Handling**: Displays clear messages when SQL errors or database connection issues occur.

Conclusion

The development of a Medical Record Management System using C++ and SQLite demonstrates how technology can streamline and secure healthcare data management. This system effectively addresses key challenges such as data security, efficient storage, and accurate handling of patient and doctor information. By leveraging SQLite's lightweight yet powerful database features alongside C++'s speed and control, the project ensures reliable and rapid data processing.

Through robust validation, error handling, and the use of prepared statements, the system maintains data integrity, prevents SQL injection attacks, and provides a foundation for scalable healthcare solutions. Ultimately, this project showcases a practical and efficient approach to digitizing medical records, improving healthcare workflows, and supporting timely clinical decision-making, paving the way for future enhancements and integrations with more advanced health information systems.