**Hotel Reservation System**

**Introduction**

The **Hotel Reservation System** project is developed using **Object-Oriented Programming (OOP)** principles in **C++**. This system simulates the essential operations of a hotel management platform by allowing users to manage different types of hotel rooms and handle reservations for specific dates. The project is designed to demonstrate key concepts of OOP, such as **abstraction**, **inheritance**, **polymorphism**, and **encapsulation**, while also focusing on **dynamic memory management** and modular software design.

1. **Assigned Task**

The objective of this project is to design and implement a Hotel Reservation System using Object-Oriented Programming (OOP) principles in C++. The system simulates the basic functionality of a hotel booking system, allowing users to manage hotel rooms (both Standard and Premium), handle reservations for specific dates, and manage room inventory dynamically.

**Functional Requirements**

**The system must provide the following features:**

• Add Room: Allow users to add new rooms of either Standard or Premium type.

• List Rooms: Display all available rooms with their reservation status.

• Reserve Room: Reserve a room for a specific date by its index.

• Remove Room: Remove a room from the hotel by its index.

• Exit: Safely terminate the program.

**Learning Objectives**

This project demonstrates core Object-Oriented Programming concepts, including:

• Abstraction: Achieved through an abstract base class RoomBase, which defines a standard interface for rooms.

• Inheritance: Different room types (StandardRoom and PremiumSuite) inherit from the base class and implement their own reservation logic.

• Polymorphism: Implemented using virtual functions that behave differently based on room type.

• Dynamic Memory Management: Rooms are managed using dynamically allocated arrays and pointers.

• Encapsulation and Modular Design: Each class handles its own data and responsibilities, promoting clarity and reusability.

1. **How the Task Was Completed**

The Hotel Reservation System was implemented using C++ and Object-Oriented Programming (OOP) principles to ensure modularity, clarity, and reusability. At its core, the system is designed around a **few key components.**

The Date structure is used to store reservation dates including day, month, and year. An abstract base class named RoomBase defines a common interface for all room types through pure virtual methods like reserve() and isReserved(), promoting abstraction.

Two classes, StandardRoom and PremiumSuite, inherit from RoomBase and provide their own logic for reserving and checking reservation status. These classes demonstrate polymorphism by offering different behavior through the same interface.

The Hotel class manages a dynamic array of room pointers, enabling addition, removal, and reservation of rooms. It uses dynamic memory allocation and deallocation to maintain flexibility while avoiding memory leaks. The program is operated via a menu-driven interface implemented in the main() function, allowing users to interactively add new rooms, remove them by index, view the list of rooms, and reserve rooms on specific dates. Input validation is carefully handled to prevent runtime errors.

Overall, the design uses encapsulation to isolate logic within classes and modularizes each function, making the system easy to maintain, scale, or extend in the future.

1. **CODE WITH COMMENTS**

#include <iostream> // Includes the standard input/output stream library

#include <limits> // Includes the numeric limits library used for input clearing

struct Date { // Defines a structure to represent a calendar date

int day, month, year; // Stores day, month, and year as integers

bool equals(const Date& other) const { // Checks if this date equals another date

return day == other.day && month == other.month && year == other.year; // Returns true if day, month, and year all match

} // End of equals function

}; // End of Date structure

class RoomBase { // Abstract base class for room types

public: // Public access specifier

virtual ~RoomBase() {} // Virtual destructor for polymorphic deletion

virtual void reserve(const Date& date) = 0; // Pure virtual function to reserve a room

virtual bool isReserved(const Date& date) const = 0; // Pure virtual function to check if a date is reserved

virtual void showReservations() const = 0; // Pure virtual function to display all reservations

}; // End of RoomBase class

class StandardRoom : public RoomBase { // Derived class representing a standard room

private: // Private access specifier

Date\* reservedDates; // Pointer to array of reserved dates

int reservedCount; // Number of dates reserved

public: // Public access specifier

StandardRoom() : reservedDates(nullptr), reservedCount(0) {} // Constructor initializing pointer to null and count to zero

~StandardRoom() { // Destructor to free dynamically allocated memory

delete[] reservedDates; // Delete the array of reserved dates

} // End of destructor

bool isReserved(const Date& date) const override { // Overrides base method to check if date is reserved

for (int i = 0; i < reservedCount; ++i) { // Loop through all reserved dates

if (reservedDates[i].equals(date)) // Check if current date matches

return true; // Return true if matched

} // End of loop

return false; // Return false if not found

} // End of isReserved

void reserve(const Date& date) override { // Overrides base method to reserve a room

if (isReserved(date)) { // If date is already reserved

std::cout << "Standard room already reserved on " // Print message

<< date.day << "/" << date.month << "/" << date.year << ".\n"; // Display date

return; // Exit function

} // End of if

Date\* newDates = new Date[reservedCount + 1]; // Create new array with space for new date

for (int i = 0; i < reservedCount; ++i) // Loop to copy old dates

newDates[i] = reservedDates[i]; // Copy each date

newDates[reservedCount] = date; // Add new date at end

delete[] reservedDates; // Delete old array

reservedDates = newDates; // Point to new array

++reservedCount; // Increment reservation count

std::cout << "Standard room successfully reserved on " // Print confirmation

<< date.day << "/" << date.month << "/" << date.year << ".\n"; // Show date

} // End of reserve

void showReservations() const override { // Overrides base method to display reservations

for (int i = 0; i < reservedCount; ++i) { // Loop through all reserved dates

std::cout << " Reserved on: " // Print reservation info

<< reservedDates[i].day << "/" // Show day

<< reservedDates[i].month << "/" // Show month

<< reservedDates[i].year << "\n"; // Show year

} // End of loop

} // End of showReservations

}; // End of StandardRoom class

class PremiumSuite : public RoomBase { // Derived class representing a premium suite

private: // Private access specifier

Date\* reservedDates; // Pointer to array of reserved dates

int reservedCount; // Number of reserved dates

public: // Public access specifier

PremiumSuite() : reservedDates(nullptr), reservedCount(0) {} // Constructor initializing pointer and count

~PremiumSuite() { // Destructor to clean up memory

delete[] reservedDates; // Delete the array of reserved dates

} // End of destructor

bool isReserved(const Date& date) const override { // Check if date is reserved

for (int i = 0; i < reservedCount; ++i) { // Loop through reservations

if (reservedDates[i].equals(date)) // Compare each date

return true; // Return true if match

} // End of loop

return false; // Return false if not found

} // End of isReserved

void reserve(const Date& date) override { // Reserve the suite for a specific date

if (isReserved(date)) { // If date already reserved

std::cout << "Premium suite already reserved on " // Show error message

<< date.day << "/" << date.month << "/" << date.year << ".\n"; // Show date

return; // Exit function

} // End of if

Date\* newDates = new Date[reservedCount + 1]; // Allocate new array

for (int i = 0; i < reservedCount; ++i) // Copy existing dates

newDates[i] = reservedDates[i]; // Copy each date

newDates[reservedCount] = date; // Add new date

delete[] reservedDates; // Delete old array

reservedDates = newDates; // Update pointer

++reservedCount; // Increase count

std::cout << "Premium suite successfully reserved on " // Print success message

<< date.day << "/" << date.month << "/" << date.year << ".\n"; // Display date

} // End of reserve

void showReservations() const override { // Show all reservations

for (int i = 0; i < reservedCount; ++i) { // Loop through all dates

std::cout << " Reserved on: " // Print reservation

<< reservedDates[i].day << "/" // Show day

<< reservedDates[i].month << "/" // Show month

<< reservedDates[i].year << "\n"; // Show year

} // End of loop

} // End of showReservations

}; // End of PremiumSuite class

class Hotel { // Represents a hotel containing multiple rooms

private: // Private access specifier

RoomBase\*\* rooms; // Pointer to dynamic array of RoomBase pointers

int roomCount; // Number of rooms

public: // Public access specifier

Hotel() : rooms(nullptr), roomCount(0) {} // Constructor initializing members

~Hotel() { // Destructor to clean up resources

for (int i = 0; i < roomCount; ++i) // Loop through rooms

delete rooms[i]; // Delete each room

delete[] rooms; // Delete array of pointers

} // End of destructor

void addRoom(RoomBase\* room) { // Adds a room to the hotel

RoomBase\*\* newRooms = new RoomBase\*[roomCount + 1]; // Allocate new array

for (int i = 0; i < roomCount; ++i) // Copy existing rooms

newRooms[i] = rooms[i]; // Copy each pointer

newRooms[roomCount++] = room; // Add new room and increment count

delete[] rooms; // Delete old array

rooms = newRooms; // Point to new array

std::cout << "Room added. Total rooms: " << roomCount << "\n"; // Confirm addition

} // End of addRoom

void removeRoom(int index) { // Removes a room by index

if (index < 0 || index >= roomCount) { // Check if index is valid

std::cout << "Invalid index.\n"; // Show error message

return; // Exit function

} // End of if

delete rooms[index]; // Delete selected room

RoomBase\*\* newRooms = new RoomBase\*[roomCount - 1]; // Allocate new array

for (int i = 0, j = 0; i < roomCount; ++i) { // Loop through all rooms

if (i != index) // Skip the removed index

newRooms[j++] = rooms[i]; // Copy remaining rooms

} // End of loop

delete[] rooms; // Delete old array

rooms = newRooms; // Update pointer

--roomCount; // Decrease count

std::cout << "Room removed. Total rooms: " << roomCount << "\n"; // Confirm removal

} // End of removeRoom

void reserveRoom(int index, const Date& date) { // Reserves a room by index

if (index < 0 || index >= roomCount) { // Check if index is valid

std::cout << "Invalid room index.\n"; // Show error

return; // Exit function

} // End of if

rooms[index]->reserve(date); // Call reserve on selected room

} // End of reserveRoom

void listRooms() const { // Lists all rooms and their reservations

for (int i = 0; i < roomCount; ++i) { // Loop through rooms

std::cout << "Room " << i << ": " // Print room index

<< (dynamic\_cast<StandardRoom\*>(rooms[i]) ? "Standard Room" : "Premium Suite") // Check room type

<< "\n"; // End line

rooms[i]->showReservations(); // Show each room's reservations

} // End of loop

} // End of listRooms

}; // End of Hotel class

void clearInput() { // Clears input buffer after a bad input

std::cin.clear(); // Clear error flags

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // Ignore the rest of the line

} // End of clearInput

int main() { // Main function

Hotel hotel; // Create a hotel object

int choice; // Variable for menu selection

while (true) { // Infinite loop for menu

std::cout << "\n--- Hotel Reservation Menu ---\n"; // Print menu header

std::cout << "0. Add Standard Room\n"; // Menu option

std::cout << "1. Add Premium Suite\n"; // Menu option

std::cout << "2. Show Room List & Reservations\n"; // Menu option

std::cout << "3. Reserve a Room\n"; // Menu option

std::cout << "4. Remove Room by Index\n"; // Menu option

std::cout << "5. Exit\n"; // Menu option

std::cout << "Enter your choice: "; // Prompt user

std::cin >> choice; // Read user choice

if (std::cin.fail()) { // Check for invalid input

clearInput(); // Clear input buffer

std::cout << "Invalid input. Try again.\n"; // Show error

continue; // Continue loop

} // End of if

switch (choice) { // Handle menu choice

case 0: // If user chooses 0

hotel.addRoom(new StandardRoom()); // Add a standard room

break; // Break out of switch

case 1: // If user chooses 1

hotel.addRoom(new PremiumSuite()); // Add a premium suite

break; // Break out of switch

case 2: // If user chooses 2

hotel.listRooms(); // List all rooms

break; // Break out of switch

case 3: { // If user chooses 3

int index; // Room index

Date date; // Reservation date

std::cout << "Enter room index: "; // Prompt for room index

std::cin >> index; // Read index

std::cout << "Enter reservation date (day month year): "; // Prompt for date

std::cin >> date.day >> date.month >> date.year; // Read date

if (std::cin.fail()) { // If input failed

clearInput(); // Clear input buffer

std::cout << "Invalid input.\n"; // Show error

} else { // If input is valid

hotel.reserveRoom(index, date); // Reserve room

} // End of if-else

break; // Break out of switch

} // End of case 3

case 4: { // If user chooses 4

int index; // Index to remove

std::cout << "Enter room index to remove: "; // Prompt

std::cin >> index; // Read index

hotel.removeRoom(index); // Remove room

break; // Break out of switch

} // End of case 4

case 5: // If user chooses 5

std::cout << "Exiting system.\n"; // Exit message

return 0; // End program

default: // For all other values

std::cout << "Invalid choice.\n"; // Show error

} // End of switch

} // End of while loop

return 0; // Return 0 from main

} // End of main