OOP Methods Used in Hostel Management System

1. ENCAPSULATION 🔒

Private Data Members

```
class Room {
private:
    int roomNumber;        // Hidden from external access
    string roomType;       // Data hiding principle
    double pricePerNight;  // Protected internal state
    bool isOccupied;
    // ... other private members
```

Public Interface Methods

```
public:
    // Getter methods (Accessors)
    int getRoomNumber() const { return roomNumber; }
    string getRoomType() const { return roomType; }
    bool getIsOccupied() const { return isOccupied; }

    // Setter methods (Mutators)
    void setPricePerNight(double newPrice) { pricePerNight = newPrice; }
```

Benefits Demonstrated:

- **Data Protection**: Private members cannot be directly accessed
- **Controlled Access**: Only through public methods
- **Validation Opportunity**: Setters can validate data before storing

2. ABSTRACTION 🎭

Complex Operations Simplified

```
// Complex check-in process abstracted into single method call
void checkIn(string guest, string date, int numNights) {
    // Internal complexity hidden from user
    if (!isOccupied) {
        isOccupied = true;
        guestName = guest;
        checkInDate = date;
        nights = numNights;
        // Display confirmation, calculate costs, etc.
    }
}
```

High-Level Interface

```
// User doesn't need to know internal implementation
HostelManagementSystem hms;
hms.run();  // Abstracts entire system operation
```

Benefits Demonstrated:

- **Simplified Usage**: Complex operations exposed as simple method calls
- **Implementation Hiding**: Users don't need to know internal details
- **Interface Consistency**: Same method call works regardless of internal changes

3. CONSTRUCTOR OVERLOADING & INITIALIZATION 🏗️

Parameterized Constructors

```
// Room constructor with parameters
Room(int num, string type, double price)
    : roomNumber(num), roomType(type), pricePerNight(price),
      isOccupied(false), guestName(""), checkInDate(""), nights(0) {}

// Guest constructor with default parameter
Guest(int id, string n, string p, string e, string addr, int room = 0)
    : guestId(id), name(n), phone(p), email(e), address(addr), roomNumber(roo
m) {}
```

Benefits Demonstrated:

- **Member Initialization Lists**: Efficient initialization
- **Default Parameters**: Flexible object creation
- **Guaranteed Initialization**: All members properly set

4. METHOD OVERLOADING (Concept Applied) 🔄

Different Display Methods for Different Classes

```
// Room display method
void Room::display() const {
    cout << setw(8) << roomNumber << setw(15) << roomType << ...
}

// Guest display method
void Guest::display() const {
    cout << setw(5) << guestId << setw(20) << name << ...
}

// Booking display method
void Booking::display() const {
    cout << setw(8) << bookingId << setw(8) << guestId << ...
}
```

Benefits Demonstrated:

- **Same Method Name**: Different implementations for different classes
- **Context-Appropriate**: Each class displays relevant information
- **Consistent Interface**: Uniform method naming across classes

5. COMPOSITION 🧩

"Has-A" Relationships

```
class HostelManagementSystem {
private:
    vector<Room> rooms;        // HMS HAS rooms
    vector<Guest> guests;      // HMS HAS guests
    vector<Booking> bookings;  // HMS HAS bookings
```

Object Collaboration

```
// Creating booking involves multiple objects
void createBooking() {
    // Find guest object
    auto guestIt = find_if(guests.begin(), guests.end(), ...);

    // Find room object
    auto roomIt = find_if(rooms.begin(), rooms.end(), ...);

    // Create booking object using data from both
    bookings.push_back(Booking(nextBookingId++, guestId, guestIt->getName
(), ...));
}
```

Benefits Demonstrated:

- **Object Relationships**: Objects work together to achieve functionality
- **Modular Design**: Each class handles its own responsibilities

- **Reusability**: Objects can be used in different contexts

6. CONST CORRECTNESS 📏

Const Methods

```
// Getter methods marked as const - promise not to modify object
int getRoomNumber() const { return roomNumber; }
string getRoomType() const { return roomType; }
void display() const { ... }  // Display doesn't change object state
```

Const Parameters

```
// Using const references in range-based loops
for (const auto& room : rooms) {
    room.display();  // Can only call const methods
}
```

Benefits Demonstrated:

- **Immutability Guarantee**: Const methods cannot modify object
- **Compiler Enforcement**: Prevents accidental modifications
- **Interface Clarity**: Shows which methods are safe to call

7. ACCESS SPECIFIERS 🚦

Three Levels of Access Control

```
class Room {
private:          // Only this class can access
    int roomNumber;
    string roomType;

public:           // Anyone can access
    Room(int num, string type, double price);
```

```
    int getRoomNumber() const;
    void checkIn(string guest, string date, int nights);

// Note: No protected members in this code, but concept applies
// protected:      // This class and derived classes can access
};
```

Benefits Demonstrated:

- **Information Hiding**: Private members completely hidden
- **Controlled Interface**: Only public methods accessible
- **Security**: Prevents unauthorized data modification

8. OBJECT LIFECYCLE MANAGEMENT 🔄

Automatic Constructor/Destructor

```
// Constructor called when object created
HostelManagementSystem hms;  // Constructor initializes rooms, sets counter
s

// Objects stored in vectors manage their own lifecycle
rooms.push_back(Room(101, "4-Bed Dorm", 25.00));  // Room object created a
nd stored
```

State Management

```
// Objects maintain their state throughout lifecycle
void checkIn(string guest, string date, int numNights) {
    isOccupied = true;      // Change object state
    guestName = guest;      // Store data in object
    // Object remembers this information until checkout
}
```

9. AGGREGATION 📦

Container-Component Relationship

```
class HostelManagementSystem {
    vector<Room> rooms;     // HMS aggregates rooms
    vector<Guest> guests;   // HMS aggregates guests

    // Rooms and guests can exist independently
    // But are managed together by the system
};
```

Benefits Demonstrated:

- **Collection Management**: System manages multiple related objects
- **Independent Existence**: Components can exist without container
- **Unified Interface**: Single point of access to all components

10. SEPARATION OF CONCERNS 🎯

Single Responsibility Principle

```
class Room {
    // ONLY handles room-related operations
    void checkIn(...);
    void checkOut();
    void display();
};

class Guest {
    // ONLY handles guest-related data
    string getName();
    void display();
};
```

```
class HostelManagementSystem {
    // ONLY handles system coordination
    void roomManagement();
    void guestManagement();
    void run();
};
```

Benefits Demonstrated:

- **Clear Responsibilities**: Each class has specific purpose
- **Maintainability**: Changes to one class don't affect others
- **Testability**: Each component can be tested independently

11. DATA ABSTRACTION THROUGH METHODS 🎨

Business Logic Encapsulation

```
// Complex business logic hidden behind simple interface
double checkOut() {
    if (isOccupied) {
        double totalCost = pricePerNight * nights;  // Business calculation
        // Generate receipt, reset room state, etc.
        return totalCost;
    }
    return 0;
}
```

Algorithm Encapsulation

```
// Search algorithm abstracted
void searchRoom() {
    auto it = find_if(rooms.begin(), rooms.end(),
                    [roomNum](const Room& r) {
                        return r.getRoomNumber() == roomNum;
                    });
```

```
    // Complex search logic hidden from user
}
```

Summary of OOP Benefits Achieved:

✅ **Modularity**: Code organized into logical, reusable classes

✅

Maintainability: Changes isolated to specific classes

✅ **Reusability**: Objects can be used in different contexts

✅

Security: Data protection through encapsulation

✅

Abstraction: Complex operations simplified

✅

Scalability: Easy to add new features by extending classes

✅

Code Organization: Clear structure and responsibilities

✅

Error Reduction: Encapsulation prevents invalid states