Event Management System Documentation

Design Choices

1. Inheritance Utilization

Inheritance was employed to create a hierarchy of event types:

- Base Class (Event): The Event class defines common attributes and methods that all events share, including:
  - Attributes: eventName, eventDate, location, and notificationPreference.
  - Methods: getDetails(), isUpcoming(), and setNotificationPreference().
- Derived Classes (Workshop, Concert, Conference): Each derived class inherits from the Event base class and extends its functionality by adding specific attributes and methods relevant to each event type. For instance:
  - **Workshop**: Adds duration and instructor.
  - **Concert**: Adds bandName and genre.
  - **Conference**: Adds speakers and topics

  This design choice promotes code reuse and helps maintain a clean and organized codebase, allowing for easy addition of new event types in the future.

2. Use of a Vector of Pointers

The decision to use a std::vector of pointers to Event objects was made for the following reasons:

- **Polymorphism**: Using pointers allows for polymorphic behavior, enabling the system to handle different event types uniformly. When calling methods on pointers, the correct derived class method is invoked, based on the actual object type, allowing for dynamic behavior at runtime.
- **Memory Management:** Storing pointers in a std::vector allows for efficient memory management through smart pointers (std::unique_ptr). This ensures automatic cleanup of allocated memory when the vector goes out of scope, preventing memory leaks.
- **Dynamic Sizing**: The std::vector provides dynamic sizing capabilities, allowing the system to manage any number of events without needing to define a fixed size in advance.

3. Benefits of Using Design Patterns

The system employs design patterns to enhance its architecture:

- **Factory Method Pattern**: This pattern is used to create instances of event subclasses. It provides a centralized location for object creation, making it easier to add new event types in the future without modifying existing code. For instance, to introduce a new event type, one only needs to add a new case in the EventFactory without altering the event creation logic elsewhere.
- Strategy Pattern: This pattern is applied to search for events based on different criteria (e.g., by date, type, or location). By defining a family of search strategies, the system can easily switch between different search algorithms at runtime. This promotes a flexible search mechanism that can be extended by adding new strategies without modifying existing code.

4. Challenges Encountered During Development

Several challenges were encountered during the development of the Event Management System:

- **Inheritance Complexity:** Managing the complexity of multiple derived classes with different attributes required careful planning to ensure that shared behavior was adequately defined in the base class without leading to bloated or confusing class definitions.
- **Memory Management**: Initially, handling memory management with raw pointers posed risks of memory leaks or dangling pointers. Switching to smart pointers (std::unique_ptr) resolved these issues but required adjustments in the design, especially regarding ownership semantics.
- **Notification System Integration:** Integrating the notification system while ensuring it adhered to user preferences involved additional complexity. It required careful planning of how notifications would be triggered based on event dates and user-defined preferences.
- **User Interface Design**: Designing a console-based user interface that is both intuitive and informative was a challenge. Ensuring that user inputs are handled correctly (especially when dealing with input buffers) required attention to detail to prevent common pitfalls such as unexpected newline characters affecting input.