Advanced C++ Sample: Multithreaded Event-Driven Observer System with Factory Pattern, JSON Serialization & Benchmarking

High-ranking C++ keywords: C++20, advanced C++ project, multithreading in C++, observer pattern, RAII, smart pointers, exception handling, template metaprogramming, memory optimization, parallel algorithms, compile-time evaluation, factory design pattern, JSON serialization, structured bindings, std::ranges, std::jthread, lambda, shared_ptr vs unique_ptr

---

```
// Filename: advanced_event_system.cpp

#include <iostream>
#include <memory>
#include <vector>
#include <string>
#include <unordered_map>
#include <functional>
#include <fstream>
#include <sstream>
#include <thread>
#include <chrono>
#include <mutex>
#include <shared_mutex>
#include <exception>
```

```
#include <stdexcept>
#include <iomanip>
#include <atomic>
#include <ranges>
#include <algorithm>
#include <execution>
#include <cassert>

// ---- JSON-Like Serialization Struct ----
struct JsonSerializable {
    virtual std::string serialize() const = 0;
    virtual ~JsonSerializable() = default;
};

// ---- Custom Exception for Safe Error Handling ----
class EventSystemException : public std::runtime_error {
public:
    explicit EventSystemException(const std::string& msg)
        : std::runtime_error("[EventSystemException] " + msg) {}
};

// ---- Compile-time Unique Event Types ----
enum class EventType : int {
    LOGIN = 1,
    LOGOUT = 2,
    FILE_UPLOAD = 3,
    SYSTEM_ALERT = 4
};

constexpr const char* to_string(EventType type) {
    switch (type) {
        case EventType::LOGIN: return "LOGIN";
        case EventType::LOGOUT: return "LOGOUT";
        case EventType::FILE_UPLOAD: return "FILE_UPLOAD";
        case EventType::SYSTEM_ALERT: return "SYSTEM_ALERT";
        default: return "UNKNOWN";
    }
}

// ---- Event Base Class ----
class Event : public JsonSerializable {
protected:
    EventType type;
    std::string timestamp;

public:
    Event(EventType t, std::string ts)
        : type(t), timestamp(std::move(ts)) {}

    virtual ~Event() = default;
    EventType get_type() const { return type; }
    std::string get_timestamp() const { return timestamp; }
};

// ---- Derived Event Classes ----
class LoginEvent : public Event {
    std::string user;

public:
    LoginEvent(std::string user, std::string timestamp)
        : Event(EventType::LOGIN, std::move(timestamp)), user(std::move(user)) {}

    std::string serialize() const override {
        return "{\"type\":\"LOGIN\",\"user\":\"" + user + "\",\"timestamp\":\"" + timestamp + "\"}";
    }
};

class LogoutEvent : public Event {
    std::string user;

public:
    LogoutEvent(std::string user, std::string timestamp)
        : Event(EventType::LOGOUT, std::move(timestamp)), user(std::move(user)) {}
    std::string serialize() const override {
        return "{\"type\":\"LOGOUT\",\"user\":\"" + user + "\",\"timestamp\":\"" + timestamp + "\"}";
    }
};


// ---- Observer Interface ----
class EventListener {
public:
    virtual void on_event(const Event& event) = 0;
    virtual ~EventListener() = default;
};


// ---- Thread-Safe Event Dispatcher (Observer Pattern) ----
class EventDispatcher {
private:
    std::unordered_map<EventType, std::vector<std::shared_ptr<EventListener>>> listeners;
    mutable std::shared_mutex mutex;


public:
    void subscribe(EventType type, std::shared_ptr<EventListener> listener) {
        std::unique_lock lock(mutex);
        listeners[type].emplace_back(std::move(listener));
    }
    void notify(const Event& event) const {
        std::shared_lock lock(mutex);
        auto it = listeners.find(event.get_type());
        if (it != listeners.end()) {
            for (auto& listener : it->second) {
                listener->on_event(event);
            }
        }
    }
};

// ---- Factory Pattern to Create Events ----
class EventFactory {
public:
    static std::unique_ptr<Event> create_event(EventType type, const std::string& user, const std::string& timestamp) {
        switch (type) {
            case EventType::LOGIN:
                return std::make_unique<LoginEvent>(user, timestamp);
            case EventType::LOGOUT:
                return std::make_unique<LogoutEvent>(user, timestamp);
            default:
                throw EventSystemException("Unsupported event type in factory.");
        }
    }
```

```
};

// ---- Logger Listener (Implements Observer Interface) ----
class Logger : public EventListener {
public:
    void on_event(const Event& event) override {
        std::ofstream ofs("log.txt", std::ios::app);
        ofs << "[Logger] Event received: " << event.serialize() << "\n";
        ofs.close();
    }
};

// ---- Real-time Monitoring Listener ----
class ConsoleMonitor : public EventListener {
public:
    void on_event(const Event& event) override {
        std::cout << "[Monitor] " << event.serialize() << "\n";
    }
};

// ---- Benchmark Utility using RAII ----
class Timer {
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> start;
    std::string name;
```

```
public:
    explicit Timer(std::string name) : name(std::move(name)),
start(std::chrono::high_resolution_clock::now()) {}

    ~Timer() {
        using namespace std::chrono;
        auto duration = high_resolution_clock::now() - start;
        std::cout << "[Benchmark] " << name << " took "
                  << duration_cast<milliseconds>(duration).count() << "ms\n";
    }
};

// ---- Structured Binding + Parallel Algorithm Demonstration ----
void process_event_stats(const std::vector<std::unique_ptr<Event>>& events) {
    Timer t("Event Statistics");

    std::unordered_map<std::string, int> type_count;

    std::for_each(std::execution::par_unseq, events.begin(), events.end(),
        [&](const auto& ev) {
            std::string type_str = to_string(ev->get_type());
            #pragma omp atomic
            ++type_count[type_str];
        });
    std::cout << "--- Event Type Frequency ---\n";
    for (const auto& [type, count] : type_count) {
        std::cout << type << ": " << count << "\n";
    }
}

// ---- Multithreaded Simulation using std::jthread (C++20) ----
void simulate_event_stream(EventDispatcher& dispatcher) {
    Timer t("Event Simulation");

    std::vector<std::jthread> threads;
    std::atomic<int> counter = 0;

    for (int i = 0; i < 4; ++i) {
        threads.emplace_back([&, i](std::stop_token token) {
            while (!token.stop_requested() && counter < 20) {
                int id = ++counter;
                auto type = (id % 2 == 0) ? EventType::LOGIN : EventType::LOGOUT;
                std::string user = "User" + std::to_string(id);
                std::string timestamp = "2025-07-01T12:" + std::to_string(id) + ":00Z";

                auto event = EventFactory::create_event(type, user, timestamp);
                dispatcher.notify(*event);
                std::this_thread::sleep_for(std::chrono::milliseconds(100));
            }
        });
    }

    for (auto& t : threads) {
        t.request_stop();
    }
}

// ---- Main Function ----
int main() {
    try {
        Timer total("Total Program Execution");

        EventDispatcher dispatcher;

        auto logger = std::make_shared<Logger>();
        auto monitor = std::make_shared<ConsoleMonitor>();

        dispatcher.subscribe(EventType::LOGIN, logger);
        dispatcher.subscribe(EventType::LOGIN, monitor);
        dispatcher.subscribe(EventType::LOGOUT, monitor);

        simulate_event_stream(dispatcher);
    }
    catch (const EventSystemException& ex) {
        std::cerr << "[ERROR] " << ex.what() << "\n";
        return 1;
    }
    catch (const std::exception& ex) {
        std::cerr << "[Unhandled Exception] " << ex.what() << "\n";
        return 1;
    }

    return 0;
}
```

Advanced C++ Sample: Multithreaded Event-Driven Observer System with Factory Pattern, JSON Serialization & Benchmarking

High-ranking C++ keywords: C++20, advanced C++ project, multithreading in C++, observer pattern, RAII, smart pointers, exception handling, template metaprogramming, memory optimization, parallel algorithms, compile-time evaluation, factory design pattern, JSON serialization, structured bindings, std::ranges, std::jthread, lambda, shared_ptr vs unique_ptr

---

```
// Filename: advanced_event_system.cpp


#include <iostream>
#include <memory>
#include <vector>
#include <string>
#include <unordered_map>
#include <functional>
#include <fstream>
```

```
#include <sstream>

#include <thread>

#include <chrono>

#include <mutex>

#include <shared_mutex>

#include <exception>

#include <stdexcept>

#include <iomanip>

#include <atomic>

#include <ranges>

#include <algorithm>

#include <execution>

#include <cassert>


// ---- JSON-Like Serialization Struct ----

struct JsonSerializable {

    virtual std::string serialize() const = 0;

    virtual ~JsonSerializable() = default;

};


// ---- Custom Exception for Safe Error Handling ----

class EventSystemException : public std::runtime_error {

public:

    explicit EventSystemException(const std::string& msg)

        : std::runtime_error("[EventSystemException] " + msg) {}
```

```
};

// ---- Compile-time Unique Event Types ----
enum class EventType : int {
    LOGIN = 1,
    LOGOUT = 2,
    FILE_UPLOAD = 3,
    SYSTEM_ALERT = 4
};

constexpr const char* to_string(EventType type) {
    switch (type) {
        case EventType::LOGIN: return "LOGIN";
        case EventType::LOGOUT: return "LOGOUT";
        case EventType::FILE_UPLOAD: return "FILE_UPLOAD";
        case EventType::SYSTEM_ALERT: return "SYSTEM_ALERT";
        default: return "UNKNOWN";
    }
}

// ---- Event Base Class ----
class Event : public JsonSerializable {
protected:
    EventType type;
    std::string timestamp;
```

```
public:
    Event(EventType t, std::string ts)
        : type(t), timestamp(std::move(ts)) {}

    virtual ~Event() = default;

    EventType get_type() const { return type; }
    std::string get_timestamp() const { return timestamp; }
};

// ---- Derived Event Classes ----
class LoginEvent : public Event {
    std::string user;

public:
    LoginEvent(std::string user, std::string timestamp)
        : Event(EventType::LOGIN, std::move(timestamp)), user(std::move(user)) {}

    std::string serialize() const override {
        return "{\"type\":\"LOGIN\",\"user\":\"" + user + "\",\"timestamp\":\"" + timestamp + "\"}";
    }
};

class LogoutEvent : public Event {
    std::string user;

public:
    LogoutEvent(std::string user, std::string timestamp)
        : Event(EventType::LOGOUT, std::move(timestamp)), user(std::move(user)) {}

    std::string serialize() const override {
        return "{\"type\":\"LOGOUT\",\"user\":\"" + user + "\",\"timestamp\":\"" + timestamp +
"\"}";
    }
};

// ---- Observer Interface ----
class EventListener {
public:
    virtual void on_event(const Event& event) = 0;
    virtual ~EventListener() = default;
};

// ---- Thread-Safe Event Dispatcher (Observer Pattern) ----
class EventDispatcher {
private:
    std::unordered_map<EventType, std::vector<std::shared_ptr<EventListener>>> listeners;
    mutable std::shared_mutex mutex;
```

```
public:
    void subscribe(EventType type, std::shared_ptr<EventListener> listener) {
        std::unique_lock lock(mutex);
        listeners[type].emplace_back(std::move(listener));
    }

    void notify(const Event& event) const {
        std::shared_lock lock(mutex);
        auto it = listeners.find(event.get_type());
        if (it != listeners.end()) {
            for (auto& listener : it->second) {
                listener->on_event(event);
            }
        }
    }
};

// ---- Factory Pattern to Create Events ----
class EventFactory {
public:
    static std::unique_ptr<Event> create_event(EventType type, const std::string& user, const std::string& timestamp) {
        switch (type) {
            case EventType::LOGIN:
                return std::make_unique<LoginEvent>(user, timestamp);
            case EventType::LOGOUT:
                return std::make_unique<LogoutEvent>(user, timestamp);
            default:
                throw EventSystemException("Unsupported event type in factory.");
        }
    }
};

// ---- Logger Listener (Implements Observer Interface) ----
class Logger : public EventListener {
public:
    void on_event(const Event& event) override {
        std::ofstream ofs("log.txt", std::ios::app);
        ofs << "[Logger] Event received: " << event.serialize() << "\n";
        ofs.close();
    }
};

// ---- Real-time Monitoring Listener ----
class ConsoleMonitor : public EventListener {
public:
    void on_event(const Event& event) override {
        std::cout << "[Monitor] " << event.serialize() << "\n";
    }
};
```

```
// ---- Benchmark Utility using RAII ----
class Timer {
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> start;
    std::string name;

public:
    explicit Timer(std::string name) : name(std::move(name)), start(std::chrono::high_resolution_clock::now()) {}

    ~Timer() {
        using namespace std::chrono;
        auto duration = high_resolution_clock::now() - start;
        std::cout << "[Benchmark] " << name << " took "
                  << duration_cast<milliseconds>(duration).count() << "ms\n";
    }
};

// ---- Structured Binding + Parallel Algorithm Demonstration ----
void process_event_stats(const std::vector<std::unique_ptr<Event>>& events) {
    Timer t("Event Statistics");

    std::unordered_map<std::string, int> type_count;
    std::for_each(std::execution::par_unseq, events.begin(), events.end(),
        [&](const auto& ev) {
            std::string type_str = to_string(ev->get_type());
            #pragma omp atomic
            ++type_count[type_str];
        });

    std::cout << "--- Event Type Frequency ---\n";
    for (const auto& [type, count] : type_count) {
        std::cout << type << ": " << count << "\n";
    }
}

// ---- Multithreaded Simulation using std::jthread (C++20) ----
void simulate_event_stream(EventDispatcher& dispatcher) {
    Timer t("Event Simulation");

    std::vector<std::jthread> threads;
    std::atomic<int> counter = 0;

    for (int i = 0; i < 4; ++i) {
        threads.emplace_back([&, i](std::stop_token token) {
            while (!token.stop_requested() && counter < 20) {
                int id = ++counter;
                auto type = (id % 2 == 0) ? EventType::LOGIN : EventType::LOGOUT;
                std::string user = "User" + std::to_string(id);
                std::string timestamp = "2025-07-01T12:" + std::to_string(id) + ":00Z";

                auto event = EventFactory::create_event(type, user, timestamp);
                dispatcher.notify(*event);
                std::this_thread::sleep_for(std::chrono::milliseconds(100));
            }
        });
    }

    for (auto& t : threads) {
        t.request_stop();
    }
}

// ---- Main Function ----
int main() {
    try {
        Timer total("Total Program Execution");

        EventDispatcher dispatcher;

        auto logger = std::make_shared<Logger>();
        auto monitor = std::make_shared<ConsoleMonitor>();
        dispatcher.subscribe(EventType::LOGIN, logger);
        dispatcher.subscribe(EventType::LOGIN, monitor);
        dispatcher.subscribe(EventType::LOGOUT, monitor);

        simulate_event_stream(dispatcher);
    }
    catch (const EventSystemException& ex) {
        std::cerr << "[ERROR] " << ex.what() << "\n";
        return 1;
    }
    catch (const std::exception& ex) {
        std::cerr << "[Unhandled Exception] " << ex.what() << "\n";
        return 1;
    }

    return 0;
}
```

Key C++ Concepts Demonstrated

Feature	Purpose
std::shared_ptr, std::unique_ptr	Memory-safe pointer management
std::jthread	Clean multithreading with stop tokens
RAII (via Timer)	Automatic benchmark cleanup
EventDispatcher	Thread-safe Observer Pattern

Feature	Purpose
EventFactory	Factory Pattern with compile-time safety
EventSystemException	Custom exception handling
structured bindings	Modern unpacking
std::execution::par_unseq	Parallel STL algorithms
constexpr, enum class	Compile-time safety