Advanced C++20 Plugin System Sample

main.cpp

```
#include <iostream>
#include <thread>
#include <chrono>
#include "PluginManager.hpp"
#include "LoggerPlugin.hpp"
#include "AnalyticsPlugin.hpp"

int main() {
    PluginManager manager;

    // Register Plugins
    manager.loadPlugin(std::make_unique<LoggerPlugin>());
    manager.loadPlugin(std::make_unique<AnalyticsPlugin>());

    // Simulate event triggers
    for (int i = 0; i < 5; ++i) {
        Event event{"onTick", {{"tick", std::to_string(i)}}};
        manager.dispatchEvent(event);
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    // Unload plugins safely
    manager.unloadAll();

    return 0;
}
```

Plugin.hpp

```
#pragma once
#include "EventDispatcher.hpp"

class Plugin {
public:
    virtual ~Plugin() = default;
    virtual std::string name() const = 0;
    virtual void onLoad(EventDispatcher& dispatcher) = 0;
    virtual void onUnload(EventDispatcher& dispatcher) = 0;
};
```

EventDispatcher.hpp

```
#pragma once
#include <string>
#include <unordered_map>
#include <vector>
#include <shared_mutex>
#include <functional>
#include <any>
#include <concepts>
#include <iostream>

struct Event {
    std::string type;
    std::unordered_map<std::string, std::string> data;
};

using EventHandler = std::function<void(const Event&)>;

class EventDispatcher {
public:
    void subscribe(const std::string& eventType, EventHandler handler) {
        std::unique_lock lock(mutex_);
        handlers[eventType].emplace_back(std::move(handler));
    }

    void unsubscribeAll(const std::string& eventType) {
        std::unique_lock lock(mutex_);
        handlers.erase(eventType);
    }

    void dispatch(const Event& event) const {
        std::shared_lock lock(mutex_);
        if (auto it = handlers.find(event.type); it != handlers.end()) {
            for (const auto& handler : it->second) {
                try {
                    handler(event);
                } catch (const std::exception& ex) {
                    std::cerr << "[ERROR] Event handler failed: " << ex.what() << '\n';
                }
            }
        }
    }

private:
    mutable std::shared_mutex mutex_;
    std::unordered_map<std::string, std::vector<EventHandler>> handlers;
};
```

PluginManager.hpp

```
#pragma once
#include "Plugin.hpp"
#include <memory>
#include <vector>
#include <mutex>
#include <iostream>
#include <ranges>

class PluginManager {
public:
    void loadPlugin(std::unique_ptr<Plugin> plugin) {
        std::lock_guard lock(mutex_);
        std::cout << "[PluginManager] Loading: " << plugin->name() << '\n';
        plugin->onLoad(dispatcher);
        plugins.push_back(std::move(plugin));
    }

    void unloadAll() {
        std::lock_guard lock(mutex_);
        for (auto& plugin : plugins | std::views::reverse) {
            std::cout << "[PluginManager] Unloading: " << plugin->name() << '\n';
            plugin->onUnload(dispatcher);
        }
        plugins.clear();
    }

    void dispatchEvent(const Event& event) {
        dispatcher.dispatch(event);
    }
private:
    std::vector<std::unique_ptr<Plugin>> plugins;
    EventDispatcher dispatcher;
    std::mutex mutex_;
};
```

LoggerPlugin.hpp

```
#pragma once
#include "Plugin.hpp"
#include <iostream>

class LoggerPlugin : public Plugin {
public:
    std::string name() const override { return "LoggerPlugin"; }

    void onLoad(EventDispatcher& dispatcher) override {
        dispatcher.subscribe("onTick", [](const Event& event) {
            std::cout << "[LoggerPlugin] Tick: " << event.data.at("tick") << '\n';
        });
    }

    void onUnload(EventDispatcher& dispatcher) override {
        dispatcher.unsubscribeAll("onTick");
        std::cout << "[LoggerPlugin] Unsubscribed from events.\n";
    }
};
```

AnalyticsPlugin.hpp

```
#pragma once
#include "Plugin.hpp"
#include <iostream>
#include <atomic>

class AnalyticsPlugin : public Plugin {
public:
    std::string name() const override { return "AnalyticsPlugin"; }

    void onLoad(EventDispatcher& dispatcher) override {
        count.store(0);
        dispatcher.subscribe("onTick", [this](const Event& event) {
            auto c = count.fetch_add(1) + 1;
            std::cout << "[AnalyticsPlugin] Processed tick #" << c << '\n';
        });
    }

    void onUnload(EventDispatcher& dispatcher) override {
        dispatcher.unsubscribeAll("onTick");
        std::cout << "[AnalyticsPlugin] Processed " << count.load() << " total ticks.\n";
    }

private:
    std::atomic<int> count{0};
};
```