Enterprise-Grade Plugin-Based Simulation Engine in Advanced Modern C++

/*

File: SimulationEngine.cpp

Description: A highly modular, thread-safe, plugin-based simulation engine demonstrating advanced Modern C++ (C++17/20) features.

Keywords: Advanced C++, Modern C++, Multithreading, Design Patterns, OOP, Templates, Memory Management, Asynchronous Programming, STL, Smart Pointers, Factory Pattern, Visitor Pattern, Strategy Pattern, Singleton

*/

```
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <map>
#include <mutex>
#include <thread>
#include <future>
#include <chrono>
#include <functional>
#include <typeindex>
#include <typeinfo>
```

```
// ----------- Logging Utility ------------
class Logger {
public:
    static Logger& getInstance() {
        static Logger instance;
        return instance;
    }
    void log(const std::string& msg) {
        std::lock_guard<std::mutex> lock(mutex_);
        std::cout << "[LOG] " << msg << std::endl;
    }
private:
    Logger() = default;
    std::mutex mutex_;
};

// ----------- Abstract Plugin Interface ------------
class IPlugin {
public:
    virtual void initialize() = 0;
    virtual void execute() = 0;
    virtual void shutdown() = 0;
    virtual ~IPlugin() = default;
};
```

```
// ----------- Plugin Factory ------------
class PluginFactory {
public:
    using Creator = std::function<std::unique_ptr<IPlugin>()>;

    static PluginFactory& getInstance() {
        static PluginFactory instance;
        return instance;
    }

    void registerPlugin(const std::string& name, Creator creator) {
        creators_[name] = creator;
    }

    std::unique_ptr<IPlugin> createPlugin(const std::string& name) {
        if (creators_.count(name)) {
            return creators_[name]();
        }
        Logger::getInstance().log("Plugin not found: " + name);
        return nullptr;
    }

private:
    std::map<std::string, Creator> creators_;
```

```
};

// ---------- Sample Plugin (Physics Module) ----------
class PhysicsPlugin : public IPlugin {
public:
    void initialize() override {
        Logger::getInstance().log("PhysicsPlugin Initialized");
    }
    void execute() override {
        Logger::getInstance().log("PhysicsPlugin Executing Physics Simulation...");
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    void shutdown() override {
        Logger::getInstance().log("PhysicsPlugin Shutdown");
    }
};

// Register Plugin Automatically
struct PhysicsPluginRegistrar {
    PhysicsPluginRegistrar() {
        PluginFactory::getInstance().registerPlugin("Physics", []() {
            return std::make_unique<PhysicsPlugin>();
        });
    }
} physicsRegistrar;
```

```
// ----------- Strategy Pattern for Simulation Modes ------------
class SimulationMode {
public:
    virtual void run() = 0;
    virtual ~SimulationMode() = default;
};

class FastMode : public SimulationMode {
public:
    void run() override {
        Logger::getInstance().log("Running in Fast Mode");
    }
};

class AccurateMode : public SimulationMode {
public:
    void run() override {
        Logger::getInstance().log("Running in Accurate Mode");
    }
};

// ----------- Visitor Pattern for Entity Processing ------------
class SimulationEntity;
```

```
class EntityVisitor {
public:
    virtual void visit(SimulationEntity& entity) = 0;
    virtual ~EntityVisitor() = default;
};

class SimulationEntity {
public:
    virtual void accept(EntityVisitor& visitor) = 0;
    virtual std::string getName() const = 0;
    virtual ~SimulationEntity() = default;
};

class CarEntity : public SimulationEntity {
public:
    void accept(EntityVisitor& visitor) override {
        visitor.visit(*this);
    }
    std::string getName() const override {
        return "Car";
    }
};

class EntityLoggerVisitor : public EntityVisitor {
public:
```

```
    void visit(SimulationEntity& entity) override {
        Logger::getInstance().log("Entity: " + entity.getName());
    }
};

// ---------- Thread Pool for Concurrent Tasks ----------
class ThreadPool {
public:
    ThreadPool(size_t threads) : stop(false) {
        for (size_t i = 0; i < threads; ++i)
            workers.emplace_back([this] {
                for (;;) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        this->condition.wait(lock, [this] {
                            return this->stop || !this->tasks.empty();
                        });
                        if (this->stop && this->tasks.empty())
                            return;
                        task = std::move(this->tasks.front());
                        this->tasks.pop_front();
                    }
                    task();
                }
```

```
        });
}

template <class F>
auto enqueue(F&& f) -> std::future<decltype(f())> {
    auto task = std::make_shared<std::packaged_task<decltype(f())()>>(std::forward<F>(f));
    std::future<decltype(f())> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        tasks.emplace_back([task]() { (*task)(); });
    }
    condition.notify_one();
    return res;
}

~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }
    condition.notify_all();
    for (auto& worker : workers)
        worker.join();
}
```

```
private:
    std::vector<std::thread> workers;
    std::deque<std::function<void()>> tasks;
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
};

// ----------- Templated Config Manager (Type-safe) ------------
class ConfigManager {
public:
    template <typename T>
    void set(const std::string& key, const T& value) {
        std::lock_guard<std::mutex> lock(mutex_);
        data_[key] = std::make_shared<Holder<T>>(value);
    }

    template <typename T>
    T get(const std::string& key) {
        std::lock_guard<std::mutex> lock(mutex_);
        return std::static_pointer_cast<Holder<T>>(data_.at(key))->value;
    }

private:
    struct Base {
```

```
        virtual ~Base() = default;
    };

    template <typename T>
    struct Holder : Base {
        Holder(const T& value) : value(value) {}
        T value;
    };

    std::map<std::string, std::shared_ptr<Base>> data_;
    std::mutex mutex_;
};

// ----------- Main Simulation Engine ------------
class SimulationEngine {
public:
    SimulationEngine()
        : threadPool(std::make_unique<ThreadPool>(4)),
config(std::make_unique<ConfigManager>()) {
        Logger::getInstance().log("SimulationEngine Constructed");
    }

    void setMode(std::unique_ptr<SimulationMode> m) {
        mode = std::move(m);
    }
```

```
void addEntity(std::unique_ptr<SimulationEntity> entity) {
    entities.push_back(std::move(entity));
}

void run() {
    mode->run();

    auto physicsPlugin = PluginFactory::getInstance().createPlugin("Physics");
    if (physicsPlugin) {
        physicsPlugin->initialize();

        auto future = threadPool->enqueue([&]() {
            physicsPlugin->execute();
        });

        EntityLoggerVisitor visitor;
        for (auto& entity : entities) {
            entity->accept(visitor);
        }

        future.get(); // Wait for plugin execution
        physicsPlugin->shutdown();
    }
}
```

```
private:
    std::unique_ptr<ThreadPool> threadPool;
    std::unique_ptr<ConfigManager> config;
    std::vector<std::unique_ptr<SimulationEntity>> entities;
    std::unique_ptr<SimulationMode> mode;
};

// ----------- Main Driver (Asynchronous, Demonstrates Futures) ------------
int main() {
    Logger::getInstance().log("Initializing Simulation");

    SimulationEngine engine;
    engine.setMode(std::make_unique<AccurateMode>());
    engine.addEntity(std::make_unique<CarEntity>());

    auto future = std::async(std::launch::async, [&engine]() {
        engine.run();
    });

    future.get(); // Wait for simulation to finish

    Logger::getInstance().log("Simulation Completed");

    return 0;
```

}