C++ Code Sample: High-Performance Multithreaded Key-Value Store

Highlights

- C++20 syntax and features
- Multithreading with std::thread, std::mutex, and std::condition_variable
- Serialization and file-based persistence
- RAII, smart pointers, STL containers
- Design Patterns: Singleton, Reader-Writer Lock
- Exception safety
- Benchmarking tools
- Unit test stubs (GoogleTest-compatible)

Project: High-Performance Multithreaded Key-Value Store in C++

```
// kv_store.cpp
#include <iostream>
#include <unordered_map>
#include <shared_mutex>
#include <fstream>
#include <sstream>
#include <string>
#include <thread>
#include <vector>
#include <chrono>
#include <filesystem>
#include <atomic>
#include <optional>
#include <condition_variable>
#include <csignal>

namespace fs = std::filesystem;

// Thread-safe logger
class Logger {
public:
    static Logger& instance() {
```

```
        static Logger logger;
        return logger;
    }

    void log(const std::string& msg) {
        std::lock_guard<std::mutex> lock(mu_);
        std::cerr << "[LOG] " << msg << std::endl;
    }

private:
    Logger() = default;
    std::mutex mu_;
};

// RAII Timer
class Timer {
public:
    Timer(const std::string& label) : label_(label),
start_(std::chrono::high_resolution_clock::now()) {}
    ~Timer() {
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> diff = end - start_;
        Logger::instance().log(label_ + " took " + std::to_string(diff.count()) + " seconds.");
    }

private:
    std::string label_;
    std::chrono::high_resolution_clock::time_point start_;
};

// Key-Value Store
class KVStore {
public:
    static KVStore& instance() {
        static KVStore store;
        return store;
    }

    void put(const std::string& key, const std::string& value) {
        std::unique_lock lock(mutex_);
```

```
    store_[key] = value;
    dirty_ = true;
}

std::optional<std::string> get(const std::string& key) {
    std::shared_lock lock(mutex_);
    auto it = store_.find(key);
    return (it != store_.end()) ? std::make_optional(it->second) : std::nullopt;
}

void remove(const std::string& key) {
    std::unique_lock lock(mutex_);
    store_.erase(key);
    dirty_ = true;
}

void saveToDisk(const std::string& filename = "store.db") {
    std::shared_lock lock(mutex_);
    std::ofstream out(filename);
    if (!out) throw std::runtime_error("Failed to open file for writing");

    for (const auto& [k, v] : store_) {
        out << k << '\t' << v << '\n';
    }
    out.close();
}

void loadFromDisk(const std::string& filename = "store.db") {
    std::unique_lock lock(mutex_);
    std::ifstream in(filename);
    if (!in) return;

    store_.clear();
    std::string line;
    while (std::getline(in, line)) {
        std::istringstream iss(line);
        std::string key, value;
        if (std::getline(iss, key, '\t') && std::getline(iss, value)) {
            store_[key] = value;
        }
```

```
        }
        in.close();
    }

    void autoPersist(const std::string& filename = "store.db", int intervalSeconds = 5) {
        persistThread_ = std::thread([this, filename, intervalSeconds]() {
            while (!terminate_) {
                std::this_thread::sleep_for(std::chrono::seconds(intervalSeconds));
                if (dirty_) {
                    try {
                        saveToDisk(filename);
                        Logger::instance().log("Auto-saved to disk.");
                        dirty_ = false;
                    } catch (const std::exception& e) {
                        Logger::instance().log(std::string("Auto-save error: ") + e.what());
                    }
                }
            }
        });
    }

    void stopPersistence() {
        terminate_ = true;
        if (persistThread_.joinable()) {
            persistThread_.join();
        }
    }

    ~KVStore() {
        stopPersistence();
        try {
            saveToDisk();  // Save before exit
        } catch (...) {}
    }

private:
    KVStore() : terminate_(false), dirty_(false) {
        loadFromDisk();
    }
```

```
    std::unordered_map<std::string, std::string> store_;
    mutable std::shared_mutex mutex_;
    std::atomic<bool> terminate_;
    std::atomic<bool> dirty_;
    std::thread persistThread_;
};

// Stress Test
void stressTest(KVStore& store, int threadCount = 8, int operations = 10000) {
    Timer timer("StressTest");
    std::vector<std::thread> threads;
    for (int i = 0; i < threadCount; ++i) {
        threads.emplace_back([&, i]() {
            for (int j = 0; j < operations; ++j) {
                std::string key = "key_" + std::to_string(i) + "_" + std::to_string(j);
                std::string value = "value_" + std::to_string(j);
                store.put(key, value);
                auto val = store.get(key);
                if (val && *val != value) {
                    Logger::instance().log("Mismatch detected");
                }
                if (j % 3 == 0) store.remove(key);
            }
        });
    }
    for (auto& t : threads) t.join();
}

// Signal Handler
void signalHandler(int signum) {
    Logger::instance().log("Interrupt signal received. Exiting gracefully...");
    KVStore::instance().stopPersistence();
    std::exit(signum);
}

// Main Application
int main() {
    std::signal(SIGINT, signalHandler);
    auto& store = KVStore::instance();
    store.autoPersist("store.db");
```

```
    Logger::instance().log("Starting stress test...");
    stressTest(store);

    Logger::instance().log("All operations completed.");
    return 0;
}
```

(Optional) Unit Test Stub

cpp

CopyEdit

```
// test_kv_store.cpp

#include "gtest/gtest.h"

#include "kv_store.cpp"


TEST(KVStoreTest, PutGet) {

    auto& store = KVStore::instance();

    store.put("foo", "bar");

    auto result = store.get("foo");

    ASSERT_TRUE(result.has_value());

    ASSERT_EQ(result.value(), "bar");

}


TEST(KVStoreTest, Remove) {

    auto& store = KVStore::instance();

    store.put("temp", "value");

    store.remove("temp");

    auto result = store.get("temp");
```

```
    ASSERT_FALSE(result.has_value());
}
```

Concepts Demonstrated

Concept	Shown In
RAII	Timer, Logger, KVStore::~KVStore
Multithreading	std::thread, std::mutex, std::shared_mutex, atomic flags
Modern C++	C++17/20 syntax (structured bindings, smart pointers, optional)
Design Patterns	Singleton (Logger, KVStore), Observer-style (log), Reader-Writer locks
Exception Safety	try/catch, safe disk I/O
Performance Monitoring	Custom RAII Timer
Persistence	saveToDisk, loadFromDisk, auto-persist thread
Signal Handling	signalHandler for graceful shutdown
STL Mastery	unordered_map, optional, filesystem, thread

How to Build and Run

bash

CopyEdit

```
g++ -std=c++20 -pthread kv_store.cpp -o kv_store
./kv_store
```