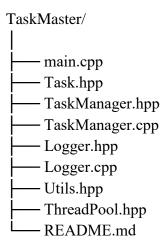
TaskMaster++ — A Scalable Task Management System in C++

This project simulates a modular, multithreaded task management system—ideal for scheduling, storing, and retrieving tasks efficiently. Think of it as a command-line lightweight Asana/Trello made in modern C++17.

File Structure



Task.hpp — Task Entity with Serialization

```
#pragma once
#include <iostream>
#include <sstream>
enum class Priority { LOW, MEDIUM, HIGH };

class Task {
  private:
    int id;
    std::string title;
    std::string description;
    Priority priority;
    bool completed;
```

```
public:
  Task(int id, const std::string& title, const std::string& desc, Priority prio)
     : id(id), title(title), description(desc), priority(prio), completed(false) {}
  int getId() const { return id; }
  std::string toJson() const {
     std::ostringstream oss;
    oss << "{ \"id\": " << id
       << ", \"title\": \"" << title
       << "\", \"description\": \"" << description
       << "\", \"priority\": \"" << (priority == Priority::HIGH ? "HIGH" : (priority ==</pre>
Priority::MEDIUM ? "MEDIUM" : "LOW"))
       << "\", \"completed\": " << (completed ? "true" : "false") << " }";</pre>
    return oss.str();
  }
  void complete() { completed = true; }
  friend std::ostream& operator<<(std::ostream& os, const Task& task) {
     os << "[Task #" << task.id << "] " << task.title
      << " | Priority: " << (task.priority == Priority::HIGH ? "HIGH" : (task.priority ==</pre>
Priority::MEDIUM ? "MEDIUM" : "LOW"))
       << " | Completed: " << (task.completed ? "Yes" : "No");</pre>
    return os;
  }
};
Logger.hpp + Logger.cpp — Singleton Logger Pattern
#pragma once
#include <fstream>
#include <mutex>
#include <string>
class Logger {
private:
  std::ofstream logFile;
  std::mutex mtx;
```

```
Logger() {
    logFile.open("log.txt", std::ios::app);
  }
  ~Logger() {
    logFile.close();
  }
public:
  Logger(const Logger&) = delete;
  Logger& operator=(const Logger&) = delete;
  static Logger& getInstance() {
    static Logger instance;
    return instance;
  }
  void log(const std::string& message) {
    std::lock_guard<std::mutex> lock(mtx);
    logFile << "[LOG] " << message << "\n";
  }
};
TaskManager.hpp + TaskManager.cpp — OOP + Smart Pointers + Factory Pattern
#pragma once
#include "Task.hpp"
#include "Logger.hpp"
#include <vector>
#include <memory>
#include <unordered_map>
#include <mutex>
class TaskManager {
private:
  std::unordered_map<int, std::unique_ptr<Task>> tasks;
  std::mutex mtx;
  int nextId = 1;
public:
```

```
TaskManager() = default;
  Task& createTask(const std::string& title, const std::string& desc, Priority prio) {
    std::lock_guard<std::mutex> lock(mtx);
    int id = nextId++;
    auto task = std::make_unique<Task>(id, title, desc, prio);
    Task& ref = *task;
    tasks[id] = std::move(task);
    Logger::getInstance().log("Created Task ID: " + std::to_string(id));
    return ref;
  }
  void completeTask(int id) {
    std::lock_guard<std::mutex> lock(mtx);
    if (tasks.count(id)) {
       tasks[id]->complete();
       Logger::getInstance().log("Completed Task ID: " + std::to_string(id));
    }
  }
  void listTasks() const {
    for (const auto& pair : tasks) {
       std::cout << *pair.second << "\n";
    }
  }
  void exportTasks() const {
    std::ofstream file("tasks.json");
    file \ll "[\n";
    bool first = true;
    for (const auto& pair : tasks) {
       if (!first) file << ",\n";
       file << " " << pair.second->toJson();
       first = false;
    file << "\n]";
};
```

ThreadPool.hpp — Simple Multithreaded Execution (C++17)

```
#pragma once
#include <thread>
#include <vector>
#include <queue>
#include <functional>
#include <mutex>
#include <condition_variable>
class ThreadPool {
private:
  std::vector<std::thread> workers;
  std::queue<std::function<void()>> jobs;
  std::mutex queueMutex;
  std::condition_variable condition;
  bool stop = false;
public:
  ThreadPool(size_t threads = std::thread::hardware_concurrency()) {
    for (size t i = 0; i < threads; ++i) {
       workers.emplace_back([this]() {
         while (true) {
            std::function<void()> job;
            {
              std::unique_lock<std::mutex> lock(queueMutex);
              condition.wait(lock, [this]() { return stop || !jobs.empty(); });
              if (stop && jobs.empty()) return;
              job = std::move(jobs.front());
              jobs.pop();
            }
            job();
       });
    }
  }
  void enqueue(std::function<void()> task) {
    {
       std::unique_lock<std::mutex> lock(queueMutex);
```

```
jobs.push(std::move(task));
    condition.notify_one();
  }
  ~ThreadPool() {
     {
       std::unique_lock<std::mutex> lock(queueMutex);
       stop = true;
     }
    condition.notify_all();
    for (std::thread &worker : workers) worker.join();
};
main.cpp — Demo Usage
#include "TaskManager.hpp"
#include "ThreadPool.hpp"
int main() {
  TaskManager manager;
  ThreadPool pool;
  pool.enqueue([&]() {
    manager.createTask("Fix Bug #404", "Resolve critical crash", Priority::HIGH);
  });
  pool.enqueue([&]() {
    manager.createTask("Design API v2", "Plan RESTful endpoints",
Priority::MEDIUM);
  });
  pool.enqueue([&]() {
     manager.createTask("Write Tests", "Unit tests for modules", Priority::LOW);
  });
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "\n\square All Tasks:\n";
  manager.listTasks();
```

```
\label{eq:manager.completeTask} $$ manager.exportTasks(); $$ std::cout << "\n Tasks Exported to `tasksjson`\n"; return 0; $$
```