HPLog – High-Performance Asynchronous Logging System

1. Project Overview

HPLog is a high-performance, asynchronous logging system written in modern C++. It is designed for systems where logging must have **minimal performance impact**, predictable behavior, and deterministic shutdown.

Instead of writing logs synchronously (which blocks application threads), HPLog uses a **background worker thread** to handle I/O while application threads only enqueue log messages. The system relies on **RAII**, **move semantics**, and **fixed-size memory buffers** to guarantee safety and performance.

This design closely resembles logging systems used in **high-frequency trading platforms, operating system services, databases, and game engines**.

---

2. Project Goals

- Ensure logging never blocks critical application logic
- Avoid heap allocation during the logging hot path
- Guarantee safe and deterministic shutdown
- Separate log production from log consumption
- Demonstrate advanced C++ system-level design

---

3. Core C++ Concepts Demonstrated

HPLog intentionally uses C++ features that are difficult or impossible to replicate efficiently in many other languages:

- **RAII (Resource Acquisition Is Initialization)** for thread and file lifetime management
- **Move semantics** for efficient message transfer
- **Minimal-lock concurrency design**
- **Deterministic destructors** for guaranteed log flushing
- **Fixed-size memory buffers** to avoid runtime allocation
- **Asynchronous producer–consumer architecture**

---

4. Architecture Overview

HPLog follows a classic **producer–consumer model**:

- Application threads act as **producers**
- A single background thread acts as the **consumer**
- Producers enqueue log messages quickly
- The consumer writes messages to disk asynchronously

This architecture ensures predictable latency and scalability.

---

5. Project Structure

```
HPLog/
│
├── include/
│   ├── log_message.hpp    // Log message definition
│   └── logger.hpp         // Logger interface
│
├── src/
│   └── logger.cpp         // Logger implementation
│
├── examples/
│   └── demo.cpp           // Usage example
│
└── README.md
```

---

6. Log Message Design

6.1 Fixed-Size Message Buffer

Each log message stores text in a fixed-size character array:

```
char text[256];
```

This design ensures: - No dynamic memory allocation - Predictable memory usage - Cache-friendly behavior

---

6.2 Move-Only Semantics

Log messages are **non-copyable** but **movable**:

```
LogMessage(const LogMessage&) = delete;
LogMessage& operator=(const LogMessage&) = delete;
```

This prevents accidental expensive copies and enforces efficient transfer between threads.

---

7. Logger Design

7.1 RAII-Based Lifetime Management

The `Logger` object owns: - Output file handle - Background worker thread - Synchronization primitives

When the `Logger` object goes out of scope: - Logging stops - Remaining messages are flushed - The worker thread is joined safely

This guarantees deterministic cleanup.

7.2 Asynchronous Processing

The logger uses: - A thread-safe queue - A condition variable - A background worker thread

Application threads never perform file I/O.

---

8. Logging Workflow

1. Application calls `logger.log(...)`
2. Message is copied into a fixed buffer
3. Message is moved into the queue
4. Background thread wakes up
5. Log entry is written to disk
6. Application continues without blocking

---

9. Example Usage

```
Logger logger("app.log");
logger.log(LogLevel::INFO, "Application started");
logger.log(LogLevel::ERROR, "Something failed");
```

On program exit, all logs are flushed automatically.

---

10. Output Format

Each log entry is written as:

```
<log_level> : <message>
```

Where `log_level` is an integer representing the severity.

---

11. Thread Safety

- Queue access is protected by a mutex
- Lock duration is kept minimal
- Condition variable avoids busy waiting
- Shutdown sequence avoids race conditions

The design is safe and predictable under concurrent load.

---

12. Performance Characteristics

- No heap allocation during logging
- Minimal lock contention
- Asynchronous file I/O
- O(1) log enqueue complexity
- Scales well with multiple producer threads

---

13. Error Handling Philosophy

HPLog avoids: - Exceptions in the logging path - Runtime error codes - Blocking calls

Instead, it relies on: - Compile-time correctness - Deterministic shutdown via RAII

---

14. Compilation Instructions

Visual Studio (Windows)

- Set C++ standard to C++17 or newer
- Build and run using Ctrl + F5

Command Line (GCC / MinGW)

```
g++ -std=c++17 -Iinclude src/logger.cpp examples/demo.cpp -o hplog
```

---

15. Limitations

- Fixed message size
- Single output sink (file only)
- Single consumer thread

These limitations are intentional for simplicity and performance predictability.

---

16. Possible Extensions

- Lock-free ring buffer
- Thread-local logging buffers
- Multiple log sinks
- Compile-time log filtering
- Custom memory pool
- Structured logging

---

17. Educational Value

This project demonstrates:

- How real-world logging systems are built
- Proper use of RAII in concurrent systems
- Safe multi-threaded shutdown
- High-performance C++ design patterns

---

18. Conclusion

HPLog is not a general-purpose logging framework. It is a **systems-level demonstration project** showing how C++ enables deterministic, high-performance infrastructure components.

Completing this project indicates strong understanding of modern C++, concurrency, and low-level design principles.