Understanding the Logger Class Implementation in C++

This document provides a detailed explanation of the implementation of a Logger class in C++, focusing on its structure, functionality, and the design patterns employed. The Logger class is designed to facilitate logging messages to a file while ensuring thread safety and adhering to the Singleton design pattern. This overview will cover header guards, necessary includes, class declaration, and key methods, providing a comprehensive understanding of how the Logger operates.

1. Header Guards in `Logger.h`

```
#ifndef LOGGER_H
#define LOGGER_H
```

Header guards are used to prevent multiple inclusions of this header file, ensuring that the **Logger** class is only defined once during compilation. The directive **#ifndef LOGGER_H** checks if **LOGGER_H** has not been defined; if true, it proceeds to define it. This pattern helps avoid redefinition errors that can lead to compilation issues.

2. `#include` Statements

```
#include <iostream>
#include <fstream>
#include <string>
#include <mutex>
```

These libraries provide essential functionality for the logger:

- **<iostream>**: Used for outputting error messages.
- **<fstream>**: Handles file operations, such as writing to the log file.
- **<string>**: Manages text strings.
- **<mutex>**: Provides thread-safety for concurrent access to the log file.

3. Singleton Logger Class Declaration

```
class Logger {
public:
    static Logger& getInstance() {
        static Logger instance;
        return instance;
    }
```

The Logger class implements the Singleton pattern, ensuring that only one instance of **Logger** exists throughout the program. The line **static Logger instance;** creates a single instance of **Logger**. Every call to **getInstance()** returns the same instance, ensuring unified access to logging functionalities.

4. Deleting the Copy Constructor and Assignment Operator

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

These lines prevent copying or assigning the **Logger** instance, reinforcing the Singleton pattern. The declaration **Logger(const Logger&) = delete;** disallows creating a copy of **Logger**, while **Logger& operator=(const Logger&) = delete;** prevents assignment of one **Logger** instance to another, ensuring that only one instance is utilized across the application.

5. `log()` Method

```
    void log(const std::string& message) {
        std::lock_guard<std::mutex> lock(mutex_);
        logFile_ << message << std::endl;
    }
```

The **log()** method writes messages to the log file safely, even when accessed by multiple threads. The line **std::lock_guard<std::mutex> lock(mutex_);** acquires a lock on **mutex_** to prevent concurrent file access. The statement **logFile_ << message << std::endl;** writes the message to the file and adds a new line. The lock is automatically released when **lock** goes out of scope, ensuring that other threads can access the log file afterward.

6. Private Data Members

```
private:
    std::ofstream logFile_;
    std::mutex mutex_;
```

These members are accessible only within the **Logger** class:

- **std::ofstream logFile_**: Manages the output file stream to write log entries to **log.txt**.
- **std::mutex mutex_**: Ensures that only one thread can write to the file at a time, guaranteeing thread safety.

7. Private Constructor and Destructor

```
    Logger() {
        logFile_.open("log.txt", std::ios::out | std::ios::app);
        if (!logFile_.is_open()) {
            std::cerr << "Failed to open log file!" << std::endl;
        }
    }

    ~Logger() {
        if (logFile_.is_open()) {
            logFile_.close();
        }
    }
};
#endif
```

The constructor is private to restrict direct instantiation of the **Logger** class, a key part of the Singleton design. It opens **log.txt** in append mode, allowing new entries to be added to existing log entries. If the file cannot be opened, an error message is printed to **std::cerr**. The destructor ensures that **logFile_** is closed when the **Logger** instance is destroyed, helping manage resources and maintain data integrity.

This document provides a comprehensive overview of the Logger class implementation, highlighting its design choices and functionality. By adhering to the Singleton pattern and ensuring thread safety, this Logger class serves as a robust solution for logging in C++ applications.