C++ Sample: Multi-threaded Bank Account System with File Logging

```
#include <iostream>
#include <fstream>
#include <thread>
#include <mutex>
#include <memory>
#include <vector>
#include <chrono>

// Global mutex for synchronizing access to shared resources
std::mutex accountMutex, logMutex;

// Logger class for writing transaction details to a file
class Logger {
private:
    std::ofstream logFile;
public:
    Logger(const std::string& filename) {
        logFile.open(filename, std::ios::app);
        if (!logFile) {
            throw std::runtime_error("Failed to open log file!");
        }
    }

    ~Logger() {
        if (logFile.is_open()) {
            logFile.close();
        }
    }

    void logTransaction(const std::string& message) {
        std::lock_guard<std::mutex> lock(logMutex);
        logFile << message << std::endl;
        std::cout << "Logged: " << message << std::endl;
    }
};
```

```
// Base class for Bank Account
class BankAccount {
protected:
    double balance;
public:
    explicit BankAccount(double initialBalance) : balance(initialBalance) {}

    virtual void deposit(double amount) = 0;
    virtual bool withdraw(double amount) = 0;
    virtual double getBalance() const = 0;
    virtual ~BankAccount() = default;
};

// Derived class for a standard bank account
class SavingsAccount : public BankAccount {
private:
    std::string accountHolder;
    std::unique_ptr<Logger> logger;
public:
    SavingsAccount(const std::string& holder, double initialBalance, const std::string&
logFile)
        : BankAccount(initialBalance), accountHolder(holder) {
        logger = std::make_unique<Logger>(logFile);
    }

    // Depositing money with thread safety
    void deposit(double amount) override {
        std::lock_guard<std::mutex> lock(accountMutex);
        balance += amount;
        logger->logTransaction(accountHolder + " deposited $" + std::to_string(amount) +
                               " | New Balance: $" + std::to_string(balance));
    }

    // Withdrawing money with thread safety
    bool withdraw(double amount) override {
        std::lock_guard<std::mutex> lock(accountMutex);
        if (amount > balance) {
            logger->logTransaction(accountHolder + " attempted to withdraw $" +
                                   std::to_string(amount) + " | Insufficient funds!");
```

```
            return false;
        }
        balance -= amount;
        logger->logTransaction(accountHolder + " withdrew $" + std::to_string(amount) +
                               " | New Balance: $" + std::to_string(balance));
        return true;
    }

    // Get current balance
    double getBalance() const override {
        return balance;
    }
};

// Function simulating concurrent transactions
void performTransactions(SavingsAccount& account, double depositAmount, double
withdrawAmount) {
    account.deposit(depositAmount);
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Simulate delay
    account.withdraw(withdrawAmount);
}

int main() {
    try {
        SavingsAccount userAccount("John Doe", 1000.0, "transactions.log");

        // Create multiple threads to simulate concurrent transactions
        std::vector<std::thread> threads;
        for (int i = 0; i < 5; ++i) {
            threads.emplace_back(performTransactions, std::ref(userAccount), 200.0, 150.0);
        }

        // Join all threads
        for (auto& thread : threads) {
            thread.join();
        }

        // Display final balance
        std::cout << "Final Account Balance: $" << userAccount.getBalance() << std::endl;
    } catch (const std::exception& e) {
```

```
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Key Features Explained

1. Encapsulation & Inheritance
   - BankAccount is an abstract base class defining the interface for deposit(), withdraw(), and getBalance().
   - SavingsAccount inherits from BankAccount and implements the functionality.

2. Multithreading with std::mutex
   - Multiple threads can deposit and withdraw money simultaneously.
   - A std::mutex ensures thread safety when modifying the balance.

3. Smart Pointers (std::unique_ptr)
   - Logger uses std::unique_ptr to manage the log file safely without memory leaks.

4. File I/O for Transaction Logging
   - Every transaction is logged in "transactions.log" for auditing.