Лабораторная работа №6 STL

Татарников Максим А-07-22

20 ВАРИАНТ

Задача №1

1. Сформировать из первых N чисел Фибоначчи, начиная с числа под номером K. Последовательность Фибоначчи считать начинающейся с 0, 1.

Задача №3

## Вариант 9. Задача «Простая бухгалтерия»

#### Входные данные

На вход программы поступает файл в формате:

Дата Счёт1 Счёт2 Сумма Примечание

Поля разделены пробелами или символами табуляции. Дата имеет формат YYYY-MM-DD или DD.MM.YYYY (2014-09-22, 22.09.2014). Счёт1 и Счёт2 – строки без пробелов. Сумма — число. Примечание – любой текст, он игнорируется.

Каждая строка отражает перевод суммы с одного счёта на другой (транзакция).

Пример файла:

01.01.2014 Старт Кошелёк 990 Изначально в кошельке было 990 руб

01.01.2014 Старт Карта 7000 А на карте 7000

20.09.2014 Кошелёк Проезд 70 Заплатил за проезд

20.09.2014 Кошелёк Базар 500 Купил фруктов на базаре

2014-09-21 Карта Книги 770.31 Купил книжку по карточке

21.02.2014 Карта Кошелёк 1000 Снял тысячу в банкомате

Файл может содержать пустые и невалидные строки. Программа должна корректно это обрабатывать.

#### *Задача*

Программа должна поддерживать два режима:

1. Вывести для всех счетов состояние на текущий момент (после обработки всех транзакций). Например, в кошельке должно остаться 990-70-500+1000 = 1420.
2. Для произвольного счёта и заданного периода времени (две даты) вычислить входящий остаток (сумму на счёте до начала периода), приход (все поступления на счёт в периоде), расход (все списания со счёта в периоде) и исходящий остаток (входящий + приход - расход).

1. STL (Standard Template Library):

1. `#include <iostream>`:

- Используется для стандартного ввода/вывода данных. Встречается везде по коду, т.к. используется для взаимодействия с пользователем через консольный ввод/вывод.

2. `#include <fstream>`:

- Используется для работы с файлами, в частности для чтения транзакций из файла и, возможно, для других операций в вашем приложении, требующих файловых операций.

3. `#include <sstream>`:

- Используется для работы с потоками строк, в основном для парсинга строк данных, таких как чтение транзакций из файла и извлечение различных полей из строк.

4. `#include <vector>`:

- Используется для хранения и управления динамическими массивами объектов типа `Transaction`. Класс `AccountManager` также использует `vector` для хранения транзакций.

5. `#include <map>`:

- Используется для хранения пар "ключ-значение", в данном случае для хранения информации о счетах (ключ - название счета, значение - баланс). Класс `AccountManager` использует `map<string, double>` для хранения счетов и их балансов.

6. `#include <algorithm>`:

- Возможно, используется для выполнения различных алгоритмов на контейнерах STL. Однако в данном коде встречается, возможно, неявно либо не использовалась.

7. `#include <iomanip>`:

- Используется для управления форматированием вывода в консоль (например, установка точности вывода чисел с плавающей запятой).

8. `#include <ctime>`:

- Используется для работы со временем и датой. Здесь используется для разбора и форматирования дат в классе `TransactionProcessor`.

9. `#include <cstdlib>`:

- В данном коде не использовался явно, возможно, был включен для использования других функций, связанных с преобразованием чисел, строк и других операций, но в самом коде не встречается

3. Принцип и алгоритм работы программы:

- Программа начинает с парсинга транзакций из файла и их обработкой.

- Затем пользователю предоставляется меню с вариантами действий (просмотр текущего состояния счетов, вычисление информации о транзакциях, выход).

- В зависимости от выбора, программа либо отображает текущее состояние счетов, либо запрашивает пользователя ввести даты и название счета, фильтрует транзакции и выводит информацию о них.

- Программа продолжает работу до тех пор, пока пользователь не выберет выход.

Описание класса `Transaction`:

Этот класс представляет отдельную транзакцию со следующими полями:

- `date` (дата),

- `accountFrom` (счёт списания),

- `accountTo` (счёт зачисления),

- `amount` (сумма),

- `comment` (комментарий).

Описание класса `AccountManager`:

Этот класс управляет счетами и их балансами. В нем есть:

- приватное поле `accounts` типа `map<string, double>`, которая хранит счета и их балансы,

- метод `processTransaction`, который обрабатывает транзакцию (уменьшает баланс со счёта списания и увеличивает баланс счёта зачисления),

- метод `printAllAccounts`, который выводит в консоль информацию о всех счетах,

- метод `getBalance`, который возвращает баланс определенного счета.

Описание класса `TransactionProcessor`:

Этот класс содержит статические методы для обработки транзакций:

- `parseTransactionsFromFile` считывает транзакции из файла и возвращает их в виде вектора,

- `filterTransactionsByAccountAndDate` фильтрует транзакции по счёту и дате,

- `isDateInRange` проверяет, входит ли дата в заданный диапазон.

Алгоритм программы

1. Начало:

- Запуск программы.

2. Парсинг и обработка транзакций:

- Считывание транзакций из файла.

- Обработка транзакций для обновления балансов счетов.

3. Отображение меню:

- Ожидание выбора пользователя.

4. Выбор действия:

- В зависимости от выбора пользователя выполнение соответствующей функциональности:

- Просмотр текущего состояния счетов.

- Вычисление информации о транзакциях для выбранного счета и периода времени.

- Завершение работы программы.

5. Просмотр текущего состояния счетов:

- Вывод информации о балансах всех счетов.

6. Ввод информации для анализа транзакций:

- Ввод названия счета, начальной даты и конечной даты.

7. Фильтрация транзакций:

- Фильтрация транзакций по выбранному счету и указанному периоду времени.

8. Вывод информации о транзакциях:

- Вывод информации о приходе, расходе и общем балансе для выбранного счета и периода времени.

9. Завершение работы:

- Выход из программы.

10. Конец.

Листинг

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

#include <algorithm>

#include <iomanip>

#include <ctime>

#include <cstdlib>

using namespace std;

class Transaction {

public:

string date;

string accountFrom;

string accountTo;

double amount;

string comment;

// Конструктор

Transaction(const string& date, const string& accountFrom, const string& accountTo,

double amount, const string& comment) :

date(date), accountFrom(accountFrom), accountTo(accountTo),

amount(amount), comment(comment) {}};

class AccountManager {

private:

map<string, double> accounts; // Используем map для хранения счетов и их балансов

public:

void processTransaction(const Transaction& transaction) {

accounts[transaction.accountFrom] -= transaction.amount; // Уменьшаем баланс счёта списания

accounts[transaction.accountTo] += transaction.amount; // Увеличиваем баланс счёта зачисления}

void printAllAccounts() const {

for (const auto& pair : accounts) {

cout << "Счет " << pair.first << ": " << pair.second << endl; }}

double getBalance(const string& account) const {

if (accounts.find(account) != accounts.end()) {

return accounts.at(account); }

else {

return 0.0; // Если счёт не найден, возвращаем 0.0};

class TransactionProcessor {

public:

static vector<Transaction> parseTransactionsFromFile(const string& filename) {

vector<Transaction> transactions;

ifstream inputFile(filename);

if (inputFile.is\_open()) {

string line;

while (getline(inputFile, line)) {

istringstream iss(line);

string date, accountFrom, accountTo, amountStr, comment;

iss >> date >> accountFrom >> accountTo >> amountStr;

getline(iss, comment); // Getting the entire comment

// Преобразование строки в число (сумму)

double amount = stod(amountStr);

// Добавление транзакции в вектор

transactions.emplace\_back(date, accountFrom, accountTo, amount, comment); }

inputFile.close();}

return transactions; }

static vector<Transaction> filterTransactionsByAccountAndDate(

const vector<Transaction>& transactions, const string& account,

const string& startDate, const string& endDate) {

vector<Transaction> filteredTransactions;

for (const auto& transaction : transactions) {

if (transaction.accountFrom == account || transaction.accountTo == account) {

if (isDateInRange(transaction.date, startDate, endDate)) {

filteredTransactions.push\_back(transaction);

}}}return filteredTransactions; }

static bool isDateInRange(const string& date, const string& startDate, const string& endDate) {

tm tm\_date = {};

istringstream ss(date);

ss >> get\_time(&tm\_date, "%Y-%m-%d"); // Trying to parse date in format "YYYY-MM-DD"

if (ss.fail()) {

ss.clear();

ss.seekg(0);

ss >> get\_time(&tm\_date, "%d.%m.%Y"); // If parsing fails, trying another format "DD.MM.YYYY"

if (ss.fail()) {

return false; // If both formats fail, the date is invalid}}

tm tm\_startDate = {};

istringstream ss\_start(startDate);

ss\_start >> get\_time(&tm\_startDate, "%Y-%m-%d");

if (ss\_start.fail()) {

ss\_start.clear();

ss\_start.seekg(0);

ss\_start >> get\_time(&tm\_startDate, "%d.%m.%Y");

if (ss\_start.fail()) {

return false; }}

tm tm\_endDate = {};

istringstream ss\_end(endDate);

ss\_end >> get\_time(&tm\_endDate, "%Y-%m-%d");

if (ss\_end.fail()) {

ss\_end.clear();

ss\_end.seekg(0);

ss\_end >> get\_time(&tm\_endDate, "%d.%m.%Y");

if (ss\_end.fail()) {

return false; }}

time\_t time\_date = mktime(&tm\_date);

time\_t time\_startDate = mktime(&tm\_startDate);

time\_t time\_endDate = mktime(&tm\_endDate);

return (time\_date >= time\_startDate && time\_date <= time\_endDate); }

static void printTransactionInfo(const vector<Transaction>& filteredTransactions, const string& account) {

double incomingBalance = 0.0;

double incomingAmount = 0.0;

double outgoingAmount = 0.0;

for (const auto& transaction : filteredTransactions) {

if (transaction.accountTo == account) {

incomingAmount += transaction.amount; }

if (transaction.accountFrom == account) {

outgoingAmount += transaction.amount; }}

cout << "Входящий остаток: " << incomingBalance << endl;

cout << "Приход: " << incomingAmount << endl;

cout << "Расход: " << outgoingAmount << endl;

cout << "Исходящий остаток: " << (incomingBalance + incomingAmount - outgoingAmount) << endl}};

string inputDate(const string& prompt) {

string date;

cout << prompt;

cin >> date;

return date; }

void printMenu() {

cout << "1. Просмотреть текущее состояние счетов" << endl;

cout << "2. Вычислить информацию о транзакциях для выбранного счёта и периода времени" << endl;

cout << "3. Выход" << endl; }

int main() {

system("chcp 1251");

setlocale(LC\_ALL, "ru");

AccountManager accountManager;

vector<Transaction> transactions = TransactionProcessor::parseTransactionsFromFile("file.txt");

// Обработка транзакций

for (const auto& transaction : transactions) {

accountManager.processTransaction(transaction); }

int choice = 1;

while (choice != 0) {

printMenu();

cin >> choice;

switch (choice) {

case 1: {

cout << "Текущее состояние счетов:" << endl;

accountManager.printAllAccounts();

break; }

case 2: {

string selectedAccount ="Кошелёк";

string startDate, endDate;

cout << "Введите название счёта: ";

cin >> selectedAccount;

cout << "Введите начальную дату (YYYY-MM-DD или DD.MM.YYYY): ";

cin >> startDate;

cout << "Введите конечную дату (YYYY-MM-DD или DD.MM.YYYY): ";

cin >> endDate;

vector<Transaction> filteredTransactions = TransactionProcessor::filterTransactionsByAccountAndDate(transactions, selectedAccount, startDate, endDate);

TransactionProcessor::printTransactionInfo(filteredTransactions, selectedAccount);break; }

case 0: {cout << "Завершение работы." << endl;break; }

default: {cout << "Неверный выбор. Попробуйте еще раз." << endl; } }}

return 0; }

Тесты

void assertTrue(bool condition, const string& testName) {

if (condition) {cout << testName << ": PASSED" << endl; }

else {cout << testName << ": FAILED" << endl; }}

// Тест конструктора

void testTransactionConstructor() {

Transaction transaction("2023-11-08", "acc1", "acc2", 100.0, "Test transaction");

assertTrue(transaction.date == "2023-11-08", "testTransactionConstructor\_date");

assertTrue(transaction.accountFrom == "acc1", "testTransactionConstructor\_accountFrom");

assertTrue(transaction.accountTo == "acc2", "testTransactionConstructor\_accountTo");

assertTrue(transaction.amount == 100.0, "testTransactionConstructor\_amount");

assertTrue(transaction.comment == "Test transaction", "testTransactionConstructor\_comment);}

// Тест метода getBalance

void testAccountManagerGetBalance() {

AccountManager accountManager;

double initialBalance = accountManager.getBalance("acc1");

assertTrue(initialBalance == 0.0, "testAccountManagerGetBalance\_initial"); // Проверка начального баланса

accountManager.processTransaction(Transaction("2023-11-08", "acc1", "acc2", 100.0, "Test transaction"));

double updatedBalance = accountManager.getBalance("acc1");

assertTrue(updatedBalance == -100.0, "testAccountManagerGetBalance\_updated"); // Проверка обновленного баланса}

// Тест метода isDateInRange

void testTransactionProcessorIsDateInRange() {

assertTrue(TransactionProcessor::isDateInRange("2023-11-08", "2023-11-01", "2023-12-01"), "testTransactionProcessorIsDateInRange\_inRange");

assertTrue(!TransactionProcessor::isDateInRange("2023-10-08", "2023-11-01", "2023-12-01"), "testTransactionProcessorIsDateInRange\_outOfRange");}

int main() {

// Запуск тестов

testTransactionConstructor();

testAccountManagerGetBalance();

testTransactionProcessorIsDateInRange();

return 0;}

Результаты:

testTransactionConstructor\_date: PASSED

testTransactionConstructor\_accountFrom: PASSED

testTransactionConstructor\_accountTo: PASSED

testTransactionConstructor\_amount: PASSED

testTransactionConstructor\_comment: PASSED

testAccountManagerGetBalance\_initial: PASSED

testAccountManagerGetBalance\_updated: PASSED

testTransactionProcessorIsDateInRange\_inRange: PASSED

testTransactionProcessorIsDateInRange\_outOfRange: PASSED