МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарные деревья

Студент гр. 9381 _____ Прибылов Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить нелинейную структуру данных — бинарное дерево, и работу с ним (ЛКП-, КЛП-, ЛПК-обходы).

Задание.

Вариант 12д

Формулу вида

< формула > ::= < терминал > | (< формула > < знак > < формула >)

< знак > ::= + | - | *

< терминал > ::= 0 | 1 | ... | 9 | *a* | *b* | ... | *z*

можно представить в виде бинарного дерева («***дерева-формулы***») с элементами типа *Elem=char* согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

- формула вида (f_1 *s* f_2) представляется деревом, в котором корень - это знак *s*, а левое и правое поддеревья - соответствующие представления формул f_1 и f_2.

- Для всех вариантов (11–17):

- для заданной формулы *f* построить дерево-формулу *t*;

- для заданного дерева-формулы *t* напечатать соответствующую формулу *f*;

Вариант 12:

- построить дерево-формулу *t* из строки, задающей формулу в префиксной форме (перечисление узлов *t* в порядке КЛП);

- преобразовать дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формуле (*f + f)*, на поддеревья, соответствующие формуле (*2 * f)*.

Основные теоретические положения.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение и ссылки на левого и правого потомка (которые могут быть и нулевыми).

<BinaryTree> ::= (<data> <BinaryTree> <BinaryTree>) | <null>

Описание алгоритмов.

1) Построение дерева по заданной арифметической формуле использует ЛКП-обход — сначала заполняется левое поддерево (путём вызова этого же алгоритма для него), затем заполняется текущий узел, затем правое поддерево (также путём вызова этого алгоритма). Условием прекращения этой рекурсии является встреченный терминальный символ — цифра или буква.

2) Печать формулы по заданному дереву использует тот же принцип — ЛКП-обход. Алгоритм печати состоит в следующем: вызывается алгоритм печати для левого поддерева (если оно непустое), затем печатается символ в текущем узле, затем вызывается алгоритм печати для правого поддерева (если непустое). Условием прекращения рекурсии являются встреченные узлы-листья.

3) Построение дерева по формуле в префиксной форме использует КЛП-обход. Сначала заполняется корень, затем запускается алгоритм: заполняется узел левого поддерева, затем вызывается этот алгоритм для левого поддерева, после чего заполняется узел правого поддерева и вызывается алгоритм для правого поддерева.

4) Преобразование формулы использует ЛПК-обход — сначала происходит спуск по дереву к листьям, где начинаются проверки и замены выражений, и происходит раскрутка обратно к корню.

Описание структур данных и функций.

template<typename T> class BinaryTree — бинарное дерево

Поля класса:

T data;

BinTree* leftTree = nullptr;

BinTree* rightTree = nullptr;

Методы класса:

explicit BinTree(T val);— конструктор класса, создаёт дерево на основе указанного выражения.

explicit BinTree(const std::string &from) — конструктор класса, принимает строку с сокращённой скобочной записью и создаёт список на основе этой строки. Если строка некорректна, создаёт пустой список.

~Hlist() — деструктор, очищает память.

bool operator==(const HList& other) const — оператор сравнения. Принимает другое дерево, возвращает результат сравнения. Проверяет идентичность деревьев, пробегая от корня до листьев, вызывая операторы сравнения пары узлов. Деревья равны, только если равны их соответствующие правые и левые потомки.

bool isLeaf() const; // проверяет, является ли текущее дерево листом (не имеет потомков)

bool isArithmeticExpression() const; — проверяет, являтся ли дерево корректным арифметическим выражений

std::string toStringPreOrder() const; — преобразует дерево в строку с арифметическим выражением в префиксной нотации, возвращает эту строку

std::string toStringInOrder() const; — преобразует дерево в строку с арифметическим выражением в классической (инфиксной) нотации

void sumToMul(); — преобразует узлы вида (f+f) в (2*f)

void readFromStringPrefix(const std::string& from, int &pos) — принимает строку и позицию, с которой нужно её читать. Создаёт дерево на основе строки с арифметическим выражением в префиксной нотации.

void readFromStringInfix(const std::string& from, int &pos) — принимает строку и позицию, с которой нужно её читать. Создаёт дерево на основе строки с арифметическим выражением в инфиксной нотации.

class Logger — вспомогательный класс для логгирования промежуточных результатов.

Методы класса:

static Logger& instance() — возвращает экземпляр класса.

void log(const std::string& str, bool toConsole = true, bool toFile = true) — принимает строку, которую нужно внести в лог, и две опции — печатать в консоль и/или в файл.

void logNodeOperatorEquals(const std::string& first, const std::string& second, bool res) — принимает две строки, соответствующие некоторым частям двух списков, и результат их сравнения для логирования.

Logger() — конструктор, создаёт файл лога и открывает его.

~Logger()— деструктор, закрывает файл лога.

Конструкторы копирования, перемещения, операторы присваивания объявлены удалёнными во избежание случайного дублирования экземпляра класса.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1	((a+a)+(a+a))	Введена строка: ((a+a)+(a+a)) Построено дерево: ((a+a)+(a+a)) Анализ: a+a Выражение заменено на 2*a Анализ: a+a Выражение заменено на 2*a Анализ: (2*a)+(2*a) Выражение заменено на 2*(2*a)

		Результат: (2*(2*a))
2	++aa+aa	Введена строка: ++aa+aa Построено дерево: ((a+a)+(a+a)) Анализ: a+a Выражение заменено на 2*a Анализ: a+a Выражение заменено на 2*a Анализ: (2*a)+(2*a) Выражение заменено на 2*(2*a) Результат: (2*(2*a))
3	((a+b)+(a+b))	ведена строка: ((a+b)+(a+b)) Построено дерево: ((a+b)+(a+b)) Анализ: a+b Выражение не изменено Анализ: a+b Выражение не изменено Анализ: (a+b)+(a+b) Выражение заменено на 2*(a+b) Результат: (2*(a+b))
4	++ab+ab	ведена строка:++ab+ab Построено дерево: ((a+b)+(a+b)) Анализ: a+b Выражение не изменено Анализ: a+b Выражение не изменено Анализ: (a+b)+(a+b) Выражение заменено на 2*(a+b) Результат: (2*(a+b))
5	((a+b)+((a+b)+(a+b)))	Введена строка: ((a+b)+((a+b)+(a+b))) Построено дерево: ((a+b)+((a+b)+(a+b))) Анализ: a+b Выражение не изменено Анализ: a+b

		Выражение не изменено Анализ: a+b Выражение не изменено Анализ: (a+b)+(a+b) Выражение заменено на 2*(a+b) Анализ: (a+b)+(2*(a+b)) Выражение не изменено Результат: ((a+b)+(2*(a+b)))
6	++ab++ab+ab	Введена строка: ++ab++ab+ab Построено дерево: ((a+b)+((a+b)+(a+b))) Анализ: a+b Выражение не изменено Анализ: a+b Выражение не изменено Анализ: a+b Выражение не изменено Анализ: (a+b)+(a+b) Выражение заменено на 2*(a+b) Анализ: (a+b)+(2*(a+b)) Выражение не изменено Результат: ((a+b)+(2*(a+b)))
7	(((d+d)+(d+d))+((d+d)+ (d+d)))	Введена строка: (((d+d)+(d+d))+((d+d)+(d+d))) Построено дерево: (((d+d)+(d+d))+((d+d)+(d+d))) Анализ: d+d Выражение заменено на 2*d Анализ: d+d Выражение заменено на 2*d Анализ: (2*d)+(2*d) Выражение заменено на 2*(2*d) Анализ: d+d Выражение заменено на 2*d Анализ: d+d Выражение заменено на 2*d Анализ: (2*d)+(2*d)

		Выражение заменено на 2*(2*d) Анализ: (2*(2*d))+(2*(2*d)) Выражение заменено на 2*(2*(2*d)) Результат: (2*(2*(2*d)))
8	+++dd+dd++dd+dd	Введена строка: +++dd+dd++dd+dd Построено дерево: (((d+d)+(d+d))+((d+d)+(d+d))) Анализ: d+d Выражение заменено на 2*d Анализ: d+d Выражение заменено на 2*d Анализ: (2*d)+(2*d) Выражение заменено на 2*(2*d) Анализ: d+d Выражение заменено на 2*d Анализ: d+d Выражение заменено на 2*d Анализ: (2*d)+(2*d) Выражение заменено на 2*(2*d) Анализ: (2*(2*d))+(2*(2*d)) Выражение заменено на 2*(2*(2*d)) Результат: (2*(2*(2*d)))
9	(((a*b)+(a*b))+((a*b)+(a*b)))	Введена строка: (((a*b)+(a*b))+((a*b)+(a*b))) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b)

		Анализ: (2*(a*b))+(2*(a*b)) Выражение заменено на 2*(2*(a*b)) Результат: (2*(2*(a*b)))
10	++*ab*ab+*ab*ab	Введена строка: ++*ab*ab+*ab*ab Построено дерево: (((a*b)+(a*b))+((a*b)+(a*b))) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: (2*(a*b))+(2*(a*b)) Выражение заменено на 2*(2*(a*b)) Результат: (2*(2*(a*b)))
11	(((a*b)+(a*b))-((a*b)+(a*b)))	Введена строка: (((a*b)+(a*b))-((a*b)+(a*b))) Построено дерево: (((a*b)+(a*b))-((a*b)+(a*b))) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b)

		Анализ: (2*(a*b))-(2*(a*b)) Выражение не изменено Результат: ((2*(a*b))-(2*(a*b)))
12	-+*ab*ab+*ab*ab	Введена строка: -+*ab*ab+*ab*ab Построено дерево: (((a*b)+(a*b))-((a*b)+(a*b))) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: a*b Выражение не изменено Анализ: a*b Выражение не изменено Анализ: (a*b)+(a*b) Выражение заменено на 2*(a*b) Анализ: (2*(a*b))-(2*(a*b)) Выражение не изменено Результат: ((2*(a*b))-(2*(a*b)))
13	a	Введена строка: a Построено дерево: a Результат: a
14	(a+a)	Введена строка: (a+a) Построено дерево: (a+a) Анализ: a+a Выражение заменено на 2*a Результат: (2*a)
15	++a	Введена строка: +aa Построено дерево: (a+a) Анализ: a+a Выражение заменено на 2*a Результат: (2*a)

16	aa	Введена строка: aa Ошибка: Дерево построено некорректно из-за ошибочного выражения.
17	+a	Введена строка: +a Ошибка: Дерево построено некорректно из-за ошибочного выражения.
18	a+	Введена строка: a+ Ошибка: Дерево построено некорректно из-за ошибочного выражения.
19	(a+)	Введена строка: (a+) Ошибка: Дерево построено некорректно из-за ошибочного выражения.
20	(+a)	Введена строка: (+a) Ошибка: Дерево построено некорректно из-за ошибочного выражения.

Выводы.

Были изучены бинарные деревья, различные методы их обхода и работа с ними.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include "BinTree.h"
#include "Logger.h"

char kDefaultStopString[] = "STOP";
char kDefaultFileName[] = "input.txt";

void printTask(); // печатает задание
void printMenu(); // печатает меню
void menu(); // вызывает меню
void consoleInput(); // организует ввод с консоли
void fileInput(); // организует ввод с файла
void performTask(std::istream& infile); // принимает поток чтения, начинает работу программы

void printTask() {
      Logger::instance().log("- Для заданной формулы f построить дерево-формулу t;\n"
                             "- Для заданного дерева-формулы t напечатать соответствующую формулу f;\n"
                          "- Построить дерево-формулу t из строки, задающей формулу в\n"
                           "  префиксной форме (перечисление узлов t в порядке КЛП);\n"
                             "- Преобразовать дерево-формулу t, заменяя в нем все поддеревья,\n"
                           "  соответствующие формуле (f + f), на поддеревья, соответствующие формуле (2 * f).\n");
}

void printMenu() {
    std::cout << "1. Ввести данные с клавиатуры.\n"
                 "2. Ввести данные с файла.\n"
                 "0. Выход из программы.\n";
}

void menu() {
    printTask();
    printMenu();
    char c = '1';
    do {
        std::cin >> c;
        std::cin.ignore(256, '\n');
        switch(c) {
            case '1':
                consoleInput();
                break;
            case '2':
                fileInput();
```

```
                    break;
                case '0':
                    std::cout << "Выход из программы.\n";
                    break;
                default:
                    std::cout << "Неверное значение.\n";
                    break;
            }
            if (c != '0') printMenu();
        } while (c != '0');
    }

    void consoleInput() {
        std::cout << "Вводите данные:\n"
                                "Чтобы вернуться в меню, введите \"" <<
kDefaultStopString << "\"\n";
        performTask(std::cin);
    }

    void fileInput() {
        std::string inputFileName;
        std::ifstream infile;
        std::cout << "Введите название файла:\n"
                            "По умолчанию данные читаются из файла \"" <<
kDefaultFileName << "\".\n";
        getline(std::cin, inputFileName);

        if (inputFileName.empty()) {
            inputFileName = kDefaultFileName;
        }

        infile.open(inputFileName);
        if (!infile) {
                std::cout << "Файла \"" << inputFileName << "\" не
существует.\n";
        } else {
              std::cout << "Чтение данных прекратится на строке \"" <<
kDefaultStopString << "\".\n";
            performTask(infile);
        }

        if (infile.is_open()) {
            infile.close();
        }
    }

    void performTask(std::istream& infile)
    {
        std::string str;

        while (!infile.eof()) {
            getline(infile, str);
            // удаляются все пробелы
                  str.erase(std::remove(str.begin(),  str.end(),  '  '),
str.end());
            if (!str.length()) continue;

            Logger::instance().log("Введена строка: " + str);
```

```
            if (str == kDefaultStopString) {
                        Logger::instance().log("Встретилась терминальная
строка.\n");
                return;
            }

            BinTree::BinTree<char> tree(str);
             // работа с деревом происходит, только если было передано
корректное выражение
                       if (tree.isArithmeticExpression() && (str ==
tree.toStringInOrder() || str == tree.toStringPreOrder())) { //
                         Logger::instance().log("Построено дерево: " +
tree.toStringInOrder()); //
                tree.sumToMul();
                                Logger::instance().log("Результат: " +
tree.toStringInOrder() + '\n');
            }
        }
    }

    int main() {
        menu();
        return 0;
    }
```

Название файла: BinTree.h

```
    #ifndef ALG_LAB3_BINTREE_H
    #define ALG_LAB3_BINTREE_H

    #include <string>
    #include <cstring>
    #include <stdexcept>
    #include "Logger.h"

    namespace BinTree {

        template<typename T>
        class BinTree {
            T data;
            BinTree* leftTree = nullptr;
            BinTree* rightTree = nullptr;

        public:
            explicit BinTree(T val);
            explicit BinTree(const std::string& str);
            ~BinTree();
            bool operator==(const BinTree& other);
            bool operator!=(const BinTree& other);

               bool isLeaf() const; // проверяет, является ли текущее
дерево листом (не имеет потомков)
                bool isArithmeticExpression() const; // проверяет, является
ли дерево корректным арифметическим выражений
              std::string toStringPreOrder() const; // преобразует дерево
в строку с арифметическим выражением в префиксной нотации
```

```
            std::string toStringInOrder() const; // преобразует дерево в строку с арифметическим выражением в классической (инфиксной) нотации
            void sumToMul(); // преобразует узлы вида (f+f) в (2*f)
        private:
            BinTree(const BinTree&) = delete;
            BinTree(BinTree&&) = delete;
            BinTree& operator=(const BinTree&) = delete;
            BinTree&& operator=(BinTree&&) = delete;
                void readFromStringPrefix(const std::string& str, int& pos); // создаёт дерево на основе строки с арифметическим выражением в префиксной нотации
            void readFromStringInfix(const std::string& str, int& pos); // создаёт дерево на основе строки с арифметическим выражением в классической (инфиксной) нотации
        };


        template<>
        bool BinTree<char>::isArithmeticExpression() const {

            if ((leftTree == nullptr) != (rightTree == nullptr))
                  return false; // дерево не может иметь только одного потомка
            if (isLeaf())
                                return  std::isalnum(static_cast<unsigned char>(data)); // в листьях могут быть только символы-терминалы
            if (!isLeaf() && !strchr("+-*", data))
                return false; // в узлах, не являющихся листьями, могут быть только +, - или *
                return leftTree->isArithmeticExpression() && rightTree->isArithmeticExpression();

        }

        template<>
           void  BinTree<char>::readFromStringPrefix(const  std::string& str, int &pos) {
            // чтение префиксной записи (КЛП)
            // текущий узел уже заполнен в предыдущем вызове метода
            // сначала заполняется левое поддерево
              leftTree = new BinTree(str.at(pos)); // заполняется узел левого поддерева
            if (!std::isalnum(static_cast<unsigned char>(str.at(pos)))) {
                  leftTree->readFromStringPrefix(str, ++pos); // затем само поддерево
                pos--;
            }
            pos++;

             rightTree = new BinTree(str.at(pos)); // заполняется узел правого поддерева
            if (!std::isalnum(static_cast<unsigned char>(str.at(pos)))) {
                  rightTree->readFromStringPrefix(str, ++pos); // затем само поддерево
                pos--;
            }
```

```
            pos++;

        }

        template<>
         void BinTree<char>::readFromStringInfix(const std::string& str,
int &pos) {
            // чтение классической (инфиксной, ЛКП) записи
            if (str.at(pos) == '(') {
                    leftTree = new BinTree('?'); // создаётся фиктивный
узел левого поддерева
                     leftTree->readFromStringInfix(str,  ++pos);  //
заполняется левое поддерево
                data = str.at(pos); // затем корень
                  rightTree = new BinTree('?'); // создаётся фиктивный
узел правого поддерева
                   rightTree->readFromStringInfix(str,  ++pos);  //
заполняется правое поддерево
                pos++;
                       }  else  if  (std::isalnum(static_cast<unsigned
char>(str.at(pos)))) {
                 data = str[pos++]; // если встретился символ-терминал,
он заносится в текущий узел
            }
        }

        template<>
        BinTree<char>::BinTree(const std::string& str) {
            try {
                int pos = 0;
                   // если строка начинается со скобки, или с символа-
терминала, выполняется чтение строки как инфиксной записи выражения
                                  if  (std::isalnum(static_cast<unsigned
char>(str.at(pos))) || str.at(pos) == '(') {
                    data = '!';
                    readFromStringInfix(str, pos);
                       // если строка начинается с символа-оператора,
выполняется чтение строки как префиксной записи выражения
                } else {
                    data = str.at(pos);
                    readFromStringPrefix(str, ++pos);
                }
                 if (!isArithmeticExpression() || pos != str.length()) {
// если формула была некорректной, конструктор сообщает об этом
                     Logger::instance().log("Ошибка: Дерево построено
некорректно из-за ошибочного выражения.\n");
                }
             } catch (std::out_of_range &) { // если произошёл выход за
границу массива - переданная формула некорректна
                    Logger::instance().log("Ошибка: Дерево построено
некорректно из-за ошибочного выражения.\n");
            }

        }

        template<typename T>
        BinTree<T>::BinTree(T val) : data(val) {}
```

```
template<typename T>
BinTree<T>::~BinTree() {
    delete leftTree;
    delete rightTree;
}

template<>
bool BinTree<char>::operator==(const BinTree<char>& other) {
     if (this->isLeaf() && other.isLeaf()) { // если оба узла - листья, сравниваются их значения
         return this->data == other.data;
     }
     if (this->isLeaf() != other.isLeaf()) { // если узлы имеют разный тип, деревья не одинаковы
         return false;
     }
     if (this->data != other.data) {
         return false;
     }
     // проверяется равенство левых и правых поддеревьев
       return *(this->leftTree) == *(other.leftTree) && *(this->rightTree) == *(other.rightTree);
}

template<>
bool BinTree<char>::operator!=(const BinTree& other) {
    return !(*this == other);
}

template<typename T>
bool BinTree<T>::isLeaf() const {
    return !leftTree && !rightTree;
}

template<>
std::string BinTree<char>::toStringPreOrder() const {
    // преобразование в префиксную (КЛП) запись
    std::string str;
    str += data; // добавляется корень
    if (leftTree)
            str += leftTree->toStringPreOrder(); // добавляется левое поддерево
    if (rightTree)
           str += rightTree->toStringPreOrder(); // добавляется правое поддерево
    return str;
}

template<>
std::string BinTree<char>::toStringInOrder() const {
    // преобразование в инфиксную (ЛКП) запись
    std::string str;
    if (!isLeaf()) str+='(';
    if (leftTree)
            str += leftTree->toStringInOrder(); // добавляется левое поддерево
    str += data; // добавляется корень
    if (rightTree)
```

```
                    str += rightTree->toStringInOrder(); // добавляется
правое поддерево
            if (!isLeaf()) str+=')';
            return str;
        }

        template<>
        void BinTree<char>::sumToMul() {
            if (isLeaf()) return;
            if (leftTree && rightTree) {
                leftTree->sumToMul();
                rightTree->sumToMul();
                        // спускается в самый низ дерева и начинает
преобразования снизу вверх
                         Logger::instance().log("Анализ: " + leftTree-
>toStringInOrder() + data + rightTree->toStringInOrder());
                  if (data == '+' && *leftTree == *rightTree) { // если
узел '+' и левое поддерево равно правому,
                      delete leftTree;                          // левое
поддерево заменяется на '2', узел - на '*'
                      leftTree = new BinTree<char>('2');
                      data = '*';
                        Logger::instance().log("Выражение заменено на " +
leftTree->toStringInOrder() + data + rightTree->toStringInOrder());
                  } else {
                      Logger::instance().log("Выражение не изменено");
                  }
            }
        }

    }
    #endif //ALG_LAB3_BINTREE_H
```

Название файла: Logger.h

```
    #ifndef ALG_LAB3_LOGGER_H
    #define ALG_LAB3_LOGGER_H

    #include <iostream>
    #include <fstream>
    #include <string>
    #include <ctime>

    class Logger {
    public:
        static Logger& instance();
         void log(const std::string& str, bool toConsole = true, bool
toFile = true);
           void logNodeOperatorEquals(const std::string& first, const
std::string& second, bool res);
    private:
        Logger();
        ~Logger();
        Logger(const Logger&) = delete;
        Logger(Logger&&) = delete;
        Logger& operator=(const Logger&) = delete;
        Logger& operator=(Logger&&) = delete;
```

```
    static Logger logger;
    std::ofstream stream;
};


#endif //ALG_LAB3_LOGGER_H
```

Название файла: Logger.cpp

```
#include "Logger.h"

Logger::Logger() {
    std::time_t t = std::time(nullptr);
    std::tm* now = std::localtime(&t);
    char logFileName[32];
    strftime(logFileName, 32, "log_%F_%T.txt", now);
    stream.open(logFileName);
}

Logger::~Logger() {
    stream.close();
}

Logger& Logger::instance() {
    static Logger instance;
    return instance;
}

void Logger::log(const std::string& str, bool toConsole, bool
toFile) {
    if (toConsole) std::cout << str << '\n';
    if (toFile) stream << str << '\n';
}
```