Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа робототехники и автоматизации

КУРСОВАЯ РАБОТА

По дисциплине «Объектно-ориентированное программирование»

Обратная Польская запись

(семестр VI)

Студент группы 3331506/20401 — И.А. Жаворонков

подпись, дата — инициалы и фамилия

Оценка выполненной студентом работы:

Преподаватель, доцент, к.ф-м.н. — М.С. Ананьевский

подпись, дата — инициалы и фамилия

Санкт-Петербург

2025

СОДЕРЖАНИЕ

Введение 3
Основная часть 4
Заключение 8
Список литературы 9
Приложение 1 10
Приложение 2 20

Введение

Обратная польская запись (далее ОПЗ) — форма записи математических и логических выражений, в которой операнды расположены перед знаками операций [1]. Такой подход устраняет необходимость использования скобок для указания порядка операций, что, в свою очередь, делает вычисления более эффективными и удобными для машинной обработки.

Для реализации вычислений с помощью ОПЗ прежде необходимо преобразовать привычную человеческому восприятию (инфиксную) запись в обратную польскую, а после провести необходимые вычисления с помощью алгоритма с использованием стека.

В рамках курсовой работы будет разработана программы на языке C++. При этом стоит учесть, что конечный код должен быть читаем и понятен, обработку ошибок необходимо реализовать через исключения, а какие-либо выводы в консоль, кроме режима отладки, запрещены.

Отдельно стоит отметить, что ОПЗ, несмотря на разработку и практическую реализацию ещё в середине XX века, остаётся актуальной и по сей день за счёт эффективного использования доступных программных ресурсов. Благодаря этому ОПЗ находит широкое применение в калькуляторах, компиляторах, стековых машинах и микроконтроллерах.

Основная часть

Как ранее уже было сказано программная реализация вычислений с помощью ОПЗ осуществляется в два этапа:

1) Преобразование инфиксной записи в обратную польскую;
2) Вычисление итогового значения выражения по ОПЗ.

Для реализации первого пункта имеется несколько вариантов, рассмотрим ниже каждый из них с точки зрения теоретического подхода для решения поставленной задачи.

1. Алгоритм «Сортировочная станция»

Этот алгоритм был разработан Эдсгером Дейкстром в 1961 году [2]. Его основная суть заключается в использовании стека для временного хранения операторов и скобок, в то время как операнды сразу добавляются в выходную строку. В этом случае выражение $A + B \times C - D$ будет представлено в постфиксной записи как $ABC \times +D -$ (см. рисунок 1).

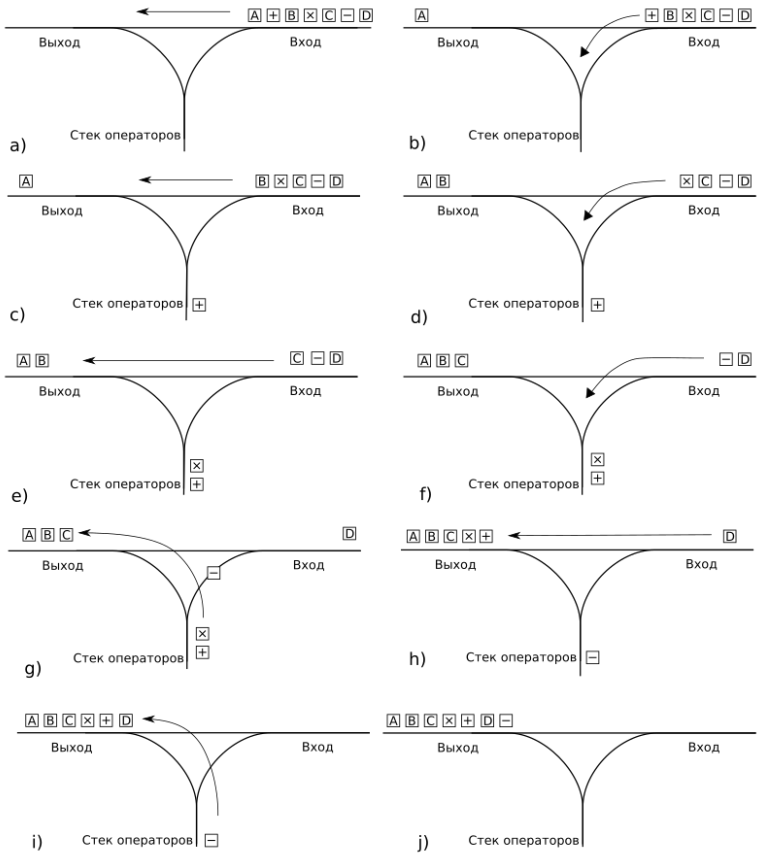

Рисунок 1 – Пример работы алгоритма «Сортировочная станция»

Значительными преимуществами алгоритма являются минимальное использование памяти, корректная обработка приоритетов и линейная сложность по времени и по памяти O(n).

2. *Рекурсивный спуск с построением синтаксического дерева*

Данный метод предполагает предварительное построение абстрактного синтаксического дерева (см. рисунок 2), по которому в дальнейшем осуществляется обход в глубину [3] для образования постфиксной записи. Такой алгоритм оказывается менее эффективным за счет затрат памяти на построение дерева. В результате сложность, по сравнению с «Сортировочной станцией» возрастает: $O_1(n) + O_2(n)$, где $O_1(n)$ – сложность построения дерева, $O_2(n)$ – сложность при его обходе.

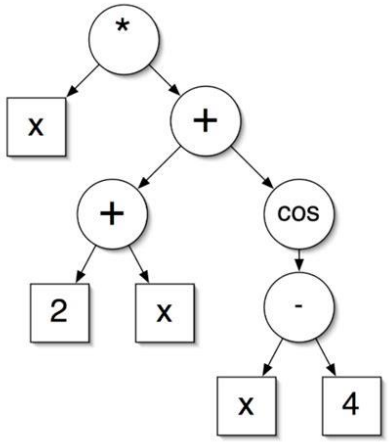

Рисунок 2 – Абстрактное синтаксическое дерево

3. *Метод двух стеков*

Этот метод во многом похож на алгоритм «Сортировочная станция», однако в процессе его реализации операнды не записываются в выходную строку, а хранятся в отдельном стеке.

В результате нетрудно сделать вывод, что оптимальным алгоритмом для поставленной задачи будет именно «Сортировочная станция» за счёт наиболее эффективного использования ресурсов.

Для вычисления значения полученной ОПЗ будем использовать стек [4] для хранения промежуточных результатов. Обработка будет осуществляться слева направо: операнды входной строки будут помещаться в стек, а при обнаружении во входных данных оператора он будет применён к верхним элементам стека. В таком случае сложность по времени и по памяти вновь будет линейной O(n), так как будет определяться одним проходом по выражению и глубиной стека в зависимости от вложенности операций соответственно.

Перейдем к практической части – реализации на C++. С учетом сформулированных ранее условий программа структурно будет разделена на три модуля:

- *Tokenizer* – модуль для преобразования входной строки в набор символов – токенов, анализируемых в дальнейшем;
- *Converter* – модуль для преобразования инфиксной записи токенов в постфиксную ОПЗ;
- *Evaluator* – вычисление значения полученной ранее ОПЗ;

Также каждый из упомянутых модулей будет содержать внутренние проверки на отсутствие ошибок, например, ввод некорректного символа, деление на нуль, неверное количество скобок. Помимо этого, дополним код модулем для генерации случайных выражений и функцией тестирования.

Код получившейся программы представлен в Приложении 1.

Для валидации решения проведем экспериментальное исследование. Для этого качественно оценим длительность выполнения программы в зависимости от количества операторов в выражении, а также в зависимости от количества выражений. Для каждого состояния программы проведем 20 тестирований для снижения влияния ошибки на результат. В итоге получим

графики зависимости скорости работы программы от длины выражений (см. рисунок 3) и от их количества (см. рисунок 4) в логарифмическом масштабе по обоим осям. Выборку по длительности выполнения программ для каждого состояния представим в Приложении 2.

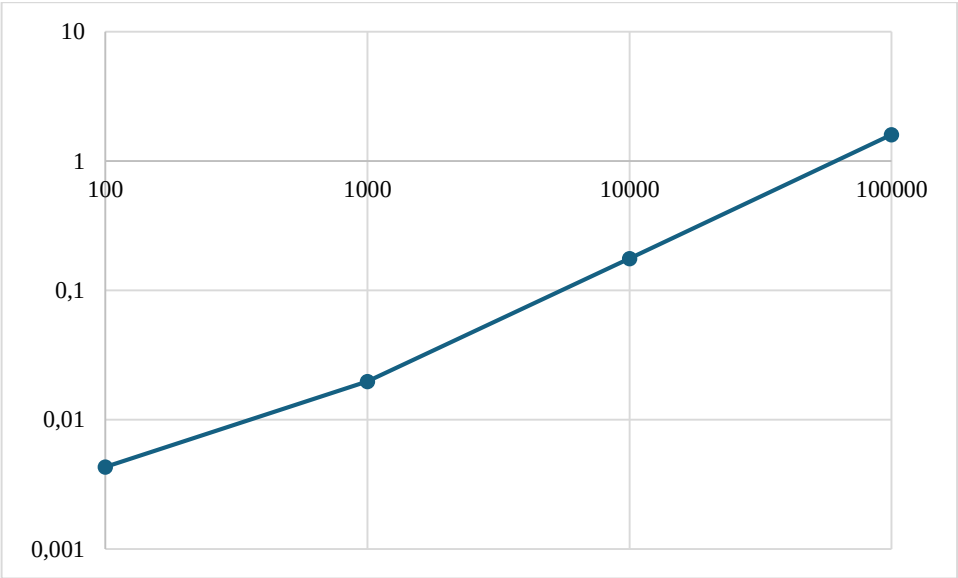

Рисунок 3 – Время выполнения программы в зависимости от количества операторов для 1000 выражений

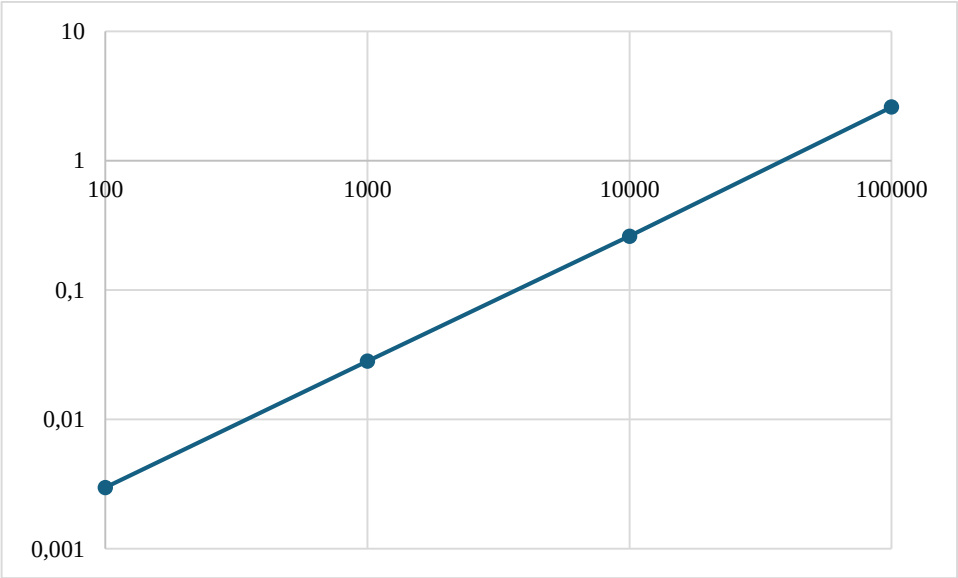

Рисунок 4 – Время выполнения программы в зависимости от количества выражений для 10 операторов

Полученные результаты подтверждают линейную сложность в обоих случаях.

Заключение

По результатам выполнения курсовой работы была достигнута поставленная ранее основная цель – реализация преобразования выражения с инфиксной записью в ОПЗ с дальнейшим вычислением конечного значения выражения.

Экспериментальное исследование подтвердило показанные в теоретической части предположения о линейности сложности при преобразовании выражений из инфиксной записи в ОПЗ и при вычислении значения выражений, записанных в ОПЗ.

Полученное время выполнения выражений с 100000 операторов составило 1,6 секунды, что, в свою очередь, является приемлемым результатом для реальных вычислений с высокой скоростью в системах с минимальными затратами по памяти. Этим и объясняется использование метода вычисления с преобразованием в ОПЗ в калькуляторах, компиляторах, микроконтроллерах и других устройствах, проводящих частые вычисления, скорость которых имеет приоритетное значение.

В качестве дальнейшего развития программы возможна реализация поддержки более сложных математический функций, таких как синус, косинус, логарифм и т.д., а также работы с переменными. Помимо этого, возможно проведение тестирования в реальных условиях и более детальный сравнительный анализ с другими методами вычислений.

В целом же проделанная работа подтвердила, что ОПЗ остаётся актуальным инструментом для обработки математических выражений за счёт своей высокой производительности, минимальных требований по памяти и относительной простоты реализации.

Список литературы

1. Обратная польская запись. Википедия. Режим доступа: https://ru.wikipedia.org/wiki/Обратная_польская_запись
2. Dijkstra, E. (1961). *Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60*. *Stichting Mathematisch Centrum. Rekenafdeling*. Stichting Mathematisch Centrum. [Электронный ресурс]. URL: https://ir.cwi.nl/pub/9251
3. Кнут, Д.Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. : Пер. с англ. – М.: ООО «И.Д. Вильямс», 2018. – 720 с. : ил. – Парал. тит. англ.
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 3rd ed. — MIT Press, 2009. — 1312 p.

Приложение 1

Код программы

```
#include <iostream>
#include <string>
#include <stack>
#include <queue>
#include <vector>
#include <stdexcept>
#include <chrono>
#include <random>
#include <sstream>
#include <cmath>
#include <iomanip>


// Типы токенов
enum class TokenType { NUMBER, OPERATOR, LEFT_PAREN, RIGHT_PAREN };


// Класс для представления токена
class Token {
public:
    TokenType type;
    double value;     // Для чисел
    char op;          // Для операторов
    bool isUnary;     // Флаг унарного оператора

    explicit Token(double val) : type(TokenType::NUMBER), value(val), isUnary(false) {}
    explicit Token(char c, bool unary = false) : type(TokenType::OPERATOR), op(c),
isUnary(unary) {
        if (c == '(') type = TokenType::LEFT_PAREN;
        else if (c == ')') type = TokenType::RIGHT_PAREN;
    }

    int getPriority() const {
        if (type == TokenType::OPERATOR) {
            if (isUnary) return 4;
            if (op == '^') return 3;
            if (op == '*' || op == '/') return 2;
            if (op == '+' || op == '-') return 1;
        }
```

```
        return 0;
    }

    bool isOperator() const { return type == TokenType::OPERATOR; }
    bool isNumber() const { return type == TokenType::NUMBER; }
    bool isLeftParenthesis() const { return type == TokenType::LEFT_PAREN; }
    bool isRightParenthesis() const { return type == TokenType::RIGHT_PAREN; }
};


// Лексический анализатор
class Tokenizer {
    std::string expr;
    size_t pos = 0;
public:
    explicit Tokenizer(const std::string& s) : expr(s) {}

    Token next() {
        while (pos < expr.size() && expr[pos] == ' ') pos++;

        if (pos >= expr.size()) throw std::invalid_argument("Unexpected end");

        // Обработка унарных операторов
        if (expr[pos] == '+' || expr[pos] == '-') {
            bool isUnary = false;

            if (pos == 0) {
                isUnary = true;
            } else {
                // Поиск предыдущего непробельного символа
                int prev = pos - 1;
                while (prev >= 0 && expr[prev] == ' ') {
                    prev--;
                }

                if (prev < 0) {
                    isUnary = true;
                } else {
                    char prev_char = expr[prev];
                    if (prev_char == '(' || prev_char == '+' || prev_char == '-' ||
                        prev_char == '*' || prev_char == '/' || prev_char == '^') {
                        isUnary = true;
                    }
```

```
            }
        }

        if (isUnary) {
            char op = expr[pos++];
            // Получаем следующий токен после унарного оператора
            Token nextToken = next();
            if (nextToken.isNumber()) {
                if (op == '-') nextToken.value = -nextToken.value;
                return nextToken;
            }
            throw std::invalid_argument("Unary operator must precede a number");
        }
    }

    if (isdigit(expr[pos]) || expr[pos] == '.') {
        size_t start = pos;
        while (pos < expr.size() && (isdigit(expr[pos]) || expr[pos] == '.')) pos++;
        return Token(stod(expr.substr(start, pos - start)));
    }

    char c = expr[pos++];
    if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^' || c == '(' || c == ')') {
        return Token(c);
    }

    throw std::invalid_argument("Invalid character: " + std::string(1, c));
  }

  bool isEnd() const { return pos >= expr.size(); }
};



// Конвертер в ОПЗ
class Converter {
  bool debug;
private:
  void printState(const std::string& msg, const Token& t, const std::stack<Token>& s, const
std::queue<Token>& q) {
      std::cout << "Step: " << msg << " " << (t.isNumber() ? std::to_string(t.value) : std::string(1,
t.op)) << std::endl;

      // Вывод стека
```

```
        std::cout << "Stack: ";
        std::stack<Token> temp_stack = s;
        std::vector<Token> stack_items;
        while (!temp_stack.empty()) {
            stack_items.push_back(temp_stack.top());
            temp_stack.pop();
        }
        for (auto it = stack_items.rbegin(); it != stack_items.rend(); ++it) {
            if (it->isNumber()) std::cout << it->value << " ";
            else std::cout << it->op << " ";
        }

        // Вывод очереди
        std::cout << "\nOutput: ";
        std::queue<Token> temp_q = q;
        while (!temp_q.empty()) {
            Token token = temp_q.front();
            temp_q.pop();
            if (token.isNumber()) std::cout << token.value << " ";
            else std::cout << token.op << " ";
        }
        std::cout << "\n\n";
    }

public:
    explicit Converter(bool d = false) : debug(d) {}

    std::queue<Token> infixToRPN(const std::string& expr) {
        std::stack<Token> op_stack;
        std::queue<Token> output;
        Tokenizer tokenizer(expr);

        while (!tokenizer.isEnd()) {
            Token token = tokenizer.next();

            if (token.isNumber()) {
                output.push(token);
                if (debug) printState("Number", token, op_stack, output);
            }
            else if (token.isLeftParenthesis()) {
                op_stack.push(token);
                if (debug) printState("(", token, op_stack, output);
            }
```

```
            else if (token.isRightParenthesis()) {
                while (!op_stack.empty() && !op_stack.top().isLeftParenthesis()) {
                    output.push(op_stack.top());
                    op_stack.pop();
                    if (debug) printState("Pop from stack", token, op_stack, output);
                }
                if (op_stack.empty()) throw std::invalid_argument("Mismatched parentheses");
                op_stack.pop(); // Удаляем '('
                if (debug) printState(")", token, op_stack, output);
            }
            else if (token.isOperator()) {
                while (!op_stack.empty() &&
                       (op_stack.top().getPriority() > token.getPriority() ||
                       (op_stack.top().getPriority() == token.getPriority() && token.op != '^')) &&
                       !op_stack.top().isLeftParenthesis()) {
                    output.push(op_stack.top());
                    op_stack.pop();
                    if (debug) printState("Pop from stack", token, op_stack, output);
                }
                op_stack.push(token);
                if (debug) printState("Operator", token, op_stack, output);
            }
        }

        while (!op_stack.empty()) {
            if (op_stack.top().isLeftParenthesis()) throw std::invalid_argument("Mismatched
parentheses");
            output.push(op_stack.top());
            op_stack.pop();
            if (debug) printState("Pop remaining", Token(' '), op_stack, output);
        }

        return output;
    }
};



// Вычислитель ОПЗ
class Evaluator {
    bool debug;
private:
    double applyOp(char op, double a, double b, bool isUnary) {
        if (isUnary) {
```

```
            switch(op) {
                case '+': return a;
                case '-': return -a;
                default: throw std::invalid_argument("Unknown unary operator");
            }
        }

        switch(op) {
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/':
                if (b == 0) throw std::invalid_argument("Division by zero");
                return a / b;
            case '^': return pow(a, b);
            default: throw std::invalid_argument("Unknown operator");
        }
    }

    void printState(const std::string& msg, const Token& t, const std::stack<double>& s) {
        std::cout << "Step: " << msg << " " << (t.isNumber() ? std::to_string(t.value) : std::string(1,
t.op)) << std::endl;
        std::cout << "Stack: ";
        std::stack<double> temp = s;
        std::vector<double> items;
        while (!temp.empty()) {
            items.push_back(temp.top());
            temp.pop();
        }
        for (auto it = items.rbegin(); it != items.rend(); ++it) {
            std::cout << *it << " ";
        }
        std::cout << "\n\n";
    }


public:
    explicit Evaluator(bool d = false) : debug(d) {}

    double evaluateRPN(std::queue<Token>& rpn) {
        std::stack<double> calc_stack;

        while (!rpn.empty()) {
```

```
            Token token = rpn.front();
            rpn.pop();

            if (token.isNumber()) {
                calc_stack.push(token.value);
                if (debug) printState("Number", token, calc_stack);
            }
            else {
                if (token.isUnary) {
                    if (calc_stack.empty()) throw std::invalid_argument("Not enough operands");
                    double a = calc_stack.top(); calc_stack.pop();
                    double result = applyOp(token.op, a, 0, true);
                    calc_stack.push(result);
                }
                else {
                    if (calc_stack.size() < 2) throw std::invalid_argument("Not enough operands");
                    double b = calc_stack.top(); calc_stack.pop();
                    double a = calc_stack.top(); calc_stack.pop();
                    double result = applyOp(token.op, a, b, false);
                    calc_stack.push(result);
                }
                if (debug) printState("Operator", token, calc_stack);
            }
        }

        if (calc_stack.size() != 1) throw std::invalid_argument("Invalid expression");
        return calc_stack.top();
    }
};


// Генератор случайных выражений
std::string generateRandomExpression(int length) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> num_dist(1, 100);
    std::uniform_int_distribution<int> op_dist(0, 5);
    std::uniform_int_distribution<int> unary_dist(0, 1);
    std::string ops = "+-*/^";

    std::stringstream ss;

    // Начало с числа или унарного оператора
```

```
    if (unary_dist(gen)) {
        ss << (unary_dist(gen) ? "+" : "-");
    }
    ss << num_dist(gen);

    for (int i = 0; i < length; ++i) {
        // Добавление оператора
        char op = ops[op_dist(gen)];
        ss << ' ' << op << ' ';

        // Возможен унарный оператор перед числом
        if (op_dist(gen) < 2 && op != '^') {
            ss << (unary_dist(gen) ? "+" : "-");
        }
        ss << num_dist(gen);

        // Возможны скобки
        if (op_dist(gen) < 1 && i < length - 2) {
            int len = std::uniform_int_distribution<int>(1, 3)(gen);
            ss << " ( ";
            if (unary_dist(gen)) ss << (unary_dist(gen) ? "+" : "-");
            ss << num_dist(gen);
            for (int j = 0; j < len; ++j) {
                ss << ' ' << ops[op_dist(gen)] << ' ';
                if (unary_dist(gen)) ss << (unary_dist(gen) ? "+" : "-");
                ss << num_dist(gen);
            }
            ss << " )";
            i += len + 2;
        }
    }

    return ss.str();
}


// Тестирование
void runTests(int count) {
    auto start = std::chrono::high_resolution_clock::now();
    int success = 0;

    for (int i = 0; i < count; ++i) {
        std::string expr = generateRandomExpression(10); // Выражения длиной 10 операторов
```

```
        try {
            Converter converter;
            Evaluator evaluator;
            std::queue<Token> rpn = converter.infixToRPN(expr);
            double result = evaluator.evaluateRPN(rpn);
            success++;

            if (i % 100 == 0) {  // Вывод каждого 100-ого выражения
                std::cout << "Expression: " << expr << "\n";
                std::cout << "Result: " << std::setprecision(6) << result << "\n\n";
            }
        } catch (const std::exception& e) {
            if (i % 100 == 0) {
                std::cout << "Failed expression: " << expr << "\n";
                std::cout << "Error: " << e.what() << "\n\n";
            }
        }
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Processed " << count << " expressions (" << success << " successful) in "
              << duration.count() << " seconds\n";
    std::cout << "Average time per expression: " << duration.count()/count << " seconds\n";
}


int main(int argc, char* argv[]) {
    if (argc > 1 && std::string(argv[1]) == "--test") {
        runTests(1000); // Тест на 1000 выражений
        return 0;
    }

    std::string expr;
    std::cout << "Enter expression: ";
    std::getline(std::cin, expr);

    try {
        bool debug = argc > 1 && std::string(argv[1]) == "--debug";

        Converter converter(debug);
        Evaluator evaluator(debug);
```

```
        std::queue<Token> rpn = converter.infixToRPN(expr);
        double result = evaluator.evaluateRPN(rpn);

        std::cout << "Result: " << result << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Приложение 2

Результаты тестов

Выражений	100	100	100	100	100	1000	10000	100000
Операторов	100	1000	10000	100000	10	10	10	10
Ср. знач., с	0.00432	0.019738	0.17605	1.59502	0.00298	0.02822	0.26072	2.60045
№ п/п	Длительность, с							
1	0.00444	0.02255	0.16939	1.61705	0.00307	0.02979	0.26117	2.59969
2	0.00430	0.02044	0.17376	1.62393	0.00303	0.02969	0.26362	2.60377
3	0.00449	0.01899	0.17258	1.60847	0.00324	0.02732	0.26073	2.60986
4	0.00439	0.01909	0.17014	1.66166	0.00329	0.02832	0.26189	2.53583
5	0.00453	0.01847	0.16769	1.56574	0.00275	0.02787	0.25299	2.57851
6	0.00406	0.01856	0.16924	1.58984	0.00322	0.02839	0.25942	2.62940
7	0.00464	0.01864	0.17712	1.56649	0.00262	0.02726	0.26101	2.60722
8	0.00431	0.01897	0.16776	1.62938	0.00285	0.02695	0.26160	2.60400
9	0.00405	0.02033	0.17198	1.55754	0.00340	0.02905	0.25671	2.58526
10	0.00491	0.02076	0.17040	1.59195	0.00289	0.02880	0.25993	2.58181
11	0.00435	0.02200	0.17554	1.55874	0.00282	0.02709	0.26027	2.59969
12	0.00425	0.01837	0.16691	1.59308	0.00294	0.02907	0.26112	2.62502
13	0.00427	0.01832	0.25732	1.56603	0.00299	0.02822	0.25850	2.61250
14	0.00423	0.01954	0.16971	1.60668	0.00276	0.03027	0.25887	2.60168
15	0.00384	0.01897	0.17560	1.56474	0.00287	0.02741	0.26331	2.61523
16	0.00441	0.02102	0.17369	1.66138	0.00280	0.02795	0.26235	2.60082
17	0.00435	0.02336	0.16912	1.56796	0.00286	0.02758	0.25950	2.60368
18	0.00445	0.01909	0.18644	1.60051	0.00251	0.02717	0.26611	2.59729
19	0.00402	0.01944	0.16994	1.56708	0.00340	0.02913	0.26294	2.61678