Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт Информационных технологий, математики и механики

Отчёт по лабораторной работе

«Вычисление арифметических выражений»

Выполнил:
студент гр. 381806-1

Сидорова А.К.

Проверил:
доцент каф. МОСТ, ИИТММ

Кустикова В.Д.

Нижний Новгород
2019 г.

Содержание

Введение 3
Постановка задачи 4
Руководство пользователя 5
Руководство программиста 7
    Описание структуры программы 7
    Описание структур данных 7
    Описание алгоритмов 9
Заключение 11
Литература 12
Приложения 13
    Приложение 1. Исходный код основной функции 13
    Приложение 2. Класс Exceptions 13
    Приложение 3. Класс TCouple 14
    Приложение 4. Класс TStack 14
    Приложение 5. Класс RPN 16

Введение

Данная лабораторная работа направлена на создание программы для преобразований арифметический выражений в обратную польскую запись и вычисления этих же выражений с помощью нее.

Обратная польская запись (*англ. Reverse Polish notation, RPN*) – форма записи арифметических и логических выражений, в которой операнды расположены перед знаками операций.

Например, арифметическому выражению $a + b * c$ будет соответствовать обратная польская запись в виде $a\ b\ c\ *\ +$.

Из-за отсутствия скобок обратная польская запись короче инфиксной, то есть обыкновенной записи, привычной многим. Из-за этого свойства при вычислениях на калькуляторах повышается скорость работы оператора, а в программируемых устройствах сокращается объём тех частей программы, которые описывают вычисления. Последнее может быть немаловажно для вычислительных устройств, которые имеют ограничения на объём используемой памяти.

Постановка задачи

В рамках данной лабораторной работы необходимо было реализовать систему для вычисления арифметических выражений путем их преобразования в обратную польскую запись.

На вход поступает арифметическое выражение в буквенном виде, то есть вместо чисел вводятся переменные. Затем пользователю отдельно необходимо ввести значения введенных переменных, которые содержатся в выражении.

На выходе пользователь сначала получает свое выражение в форме обратной польской записи, а затем список переменных, которые содержатся в этом выражении и которые необходимо инициализировать. В конечном итоге, результатом программы является значение выражения, введенного пользователем в начале работы программы.

Руководство пользователя

Рассмотрим один из вариантов использования программы, которая была разработана в ходе выполнения данной практической работы.

После запуска программы перед пользователем откроется окно (**Ошибка! Источник ссылки не найден.**), в котором будет предложено ввести арифметическое выражение, которое впоследствии будет посчитано. Стоит учесть, что необходимо ввести выражение, состоящие исключительно из однобуквенных переменных из английских символов и символов операций (‘+’, '-', '*', '/'). Кроме того, в выражении допускается использование

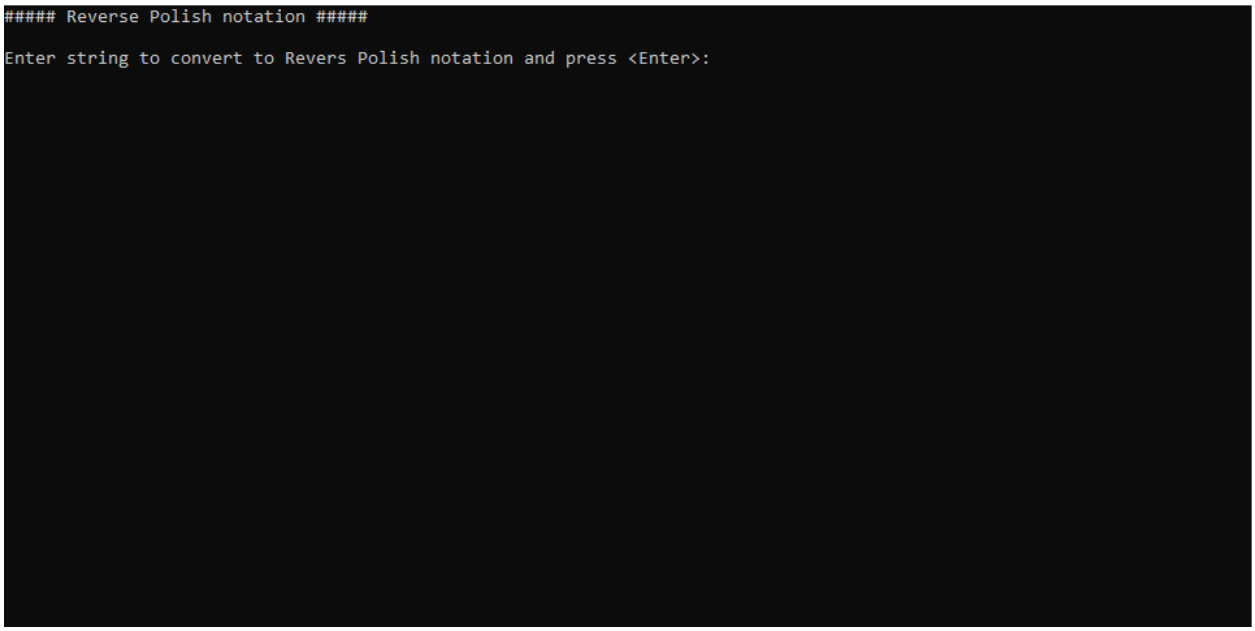

Рис. 1. *Пример работы демонстрационной программы*

пробела.

Вводим арифметическое выражение, удовлетворяющее указанным ранее правилам, и нажимаем на клавишу *<Enter>* (Пример работы демонстрационной программы).

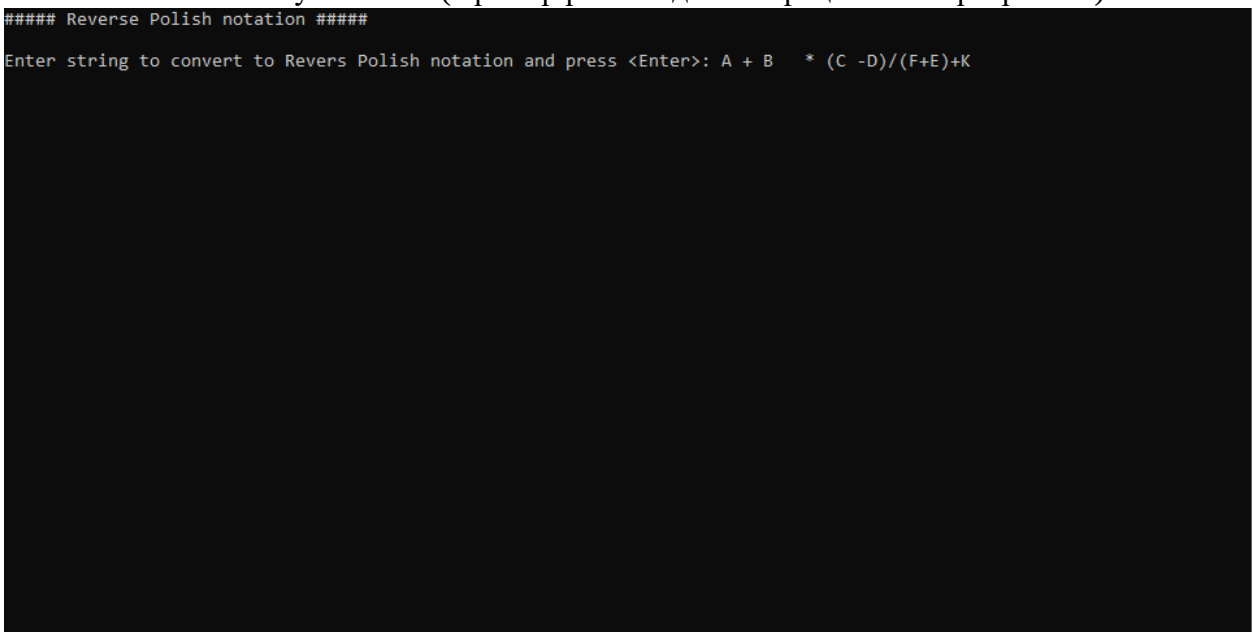

Рис. 2. *Пример работы демонстрационной программы*

Если были соблюдены все правила и указания по вводу арифметического выражения, то программа выведет напротив *«Reverse Polish Notation»* соответственно преобразованное выражение (**Ошибка! Источник ссылки не найден.**).

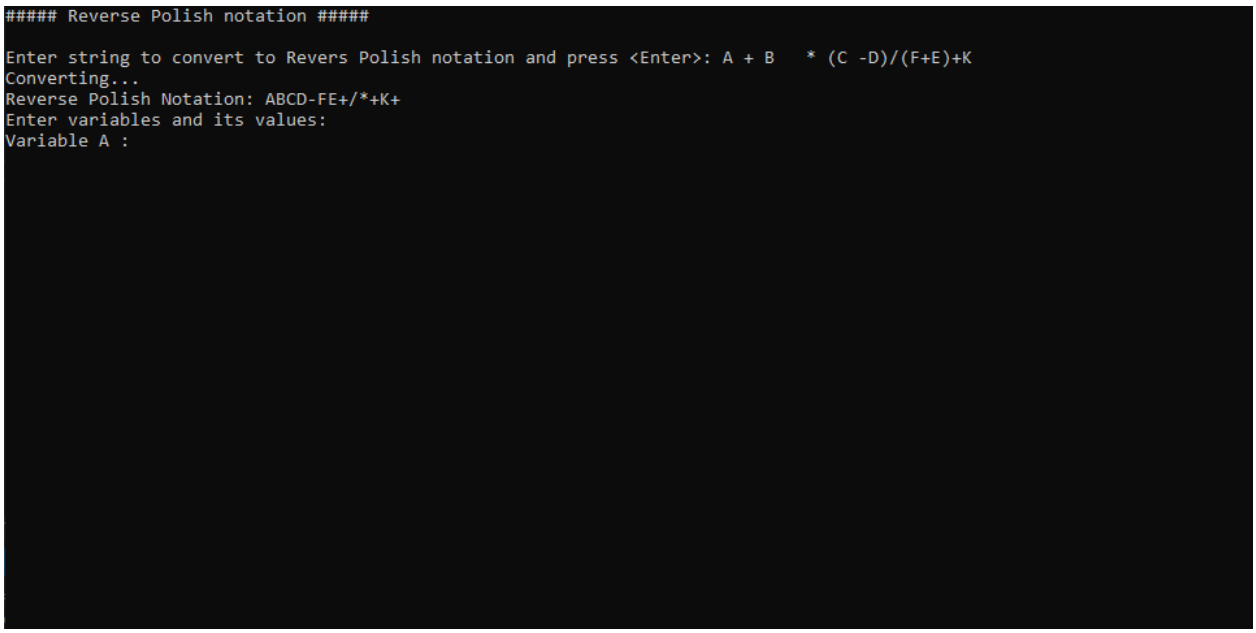

Рис. 3. *Пример работы демонстрационной программы*

После будет предложено ввести значения тех переменных, что содержатся в выражении. То есть после *«Variable <название_переменной>»* необходимо ввести то значение, которое должна принимать переменная *<название_переменной>* (оно может быть целочисленным или дробным), и нажать после ввода клавишу *<Enter>*. После инициализации последней переменной программа подсчитает выражение и выведет результат напротив «*Result*» (**Ошибка! Источник ссылки не найден.**).

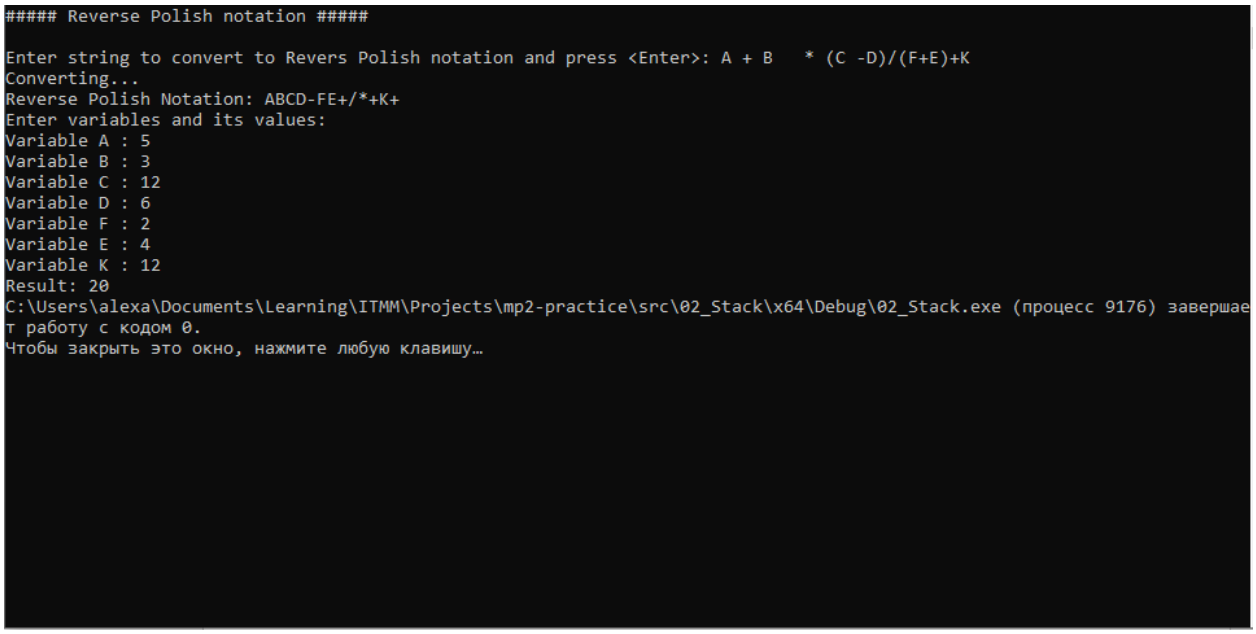

Рис. 4. *Пример работы демонстрационной программы*

Руководство программиста

Описание структуры программы

Программа состоит из следующих файлов:

- Includes\exceptions.h – заголовочный файл, который содержит в себе описание и реализацию собственного класса исключений.
- Includes\TCouple.h – заголовочный файл, который содержит в себе описание и реализацию шаблонного класса для хранения пары зависимых между собой значений – имя переменной и ее значение.
- Includes\TStack.h – заголовочный файл, в котором реализован класс стека, как структуры данных.
- Includes\RPN.h – заголовочный файл, в котором хранится статический класс RPN с методами для работы с арифметическими выражениями.
- main.cpp – исполняемый файл, в котором содержится главная функция программы.

Описание структур данных

1. **Класс Exceptions – класс исключений.**

   Класс содержит в себе:

   - `string msg` – переменная, которая хранит в себе строку с информацией об ошибке.
   - `Exception(string _msg) : msg(_msg) {}` - конструктор с параметром.
   - `const char* what() const noexcept` – константный метод, который возвращает строку, содержащую в себе информацию об ошибке.

2. **Класс TCouple – вспомогательный шаблонный класс пары значений.**

   Класс реализован для работы с парой данных: название переменной и ее значение. Содержит в себе:

   - `char var` – переменная, которая хранит в себе символ – имя переменной.
   - `ValType value` – значение переменной `var` типа `ValType`.
   - `TCouple()` – конструктор по умолчанию.

3. **Класс TStack – шаблонный класс структуры данных стек.**

   Класс представляет собой реализацию такой структуры данных, как стек. Содержит в себе:

   - `int size` – переменная, которая хранит в себе размер стека.
   - `int top` – переменная, которая хранит в себе номер последнего элемента, хранящегося в стеке.
   - `ValType* elem` – указатель на область памяти, где хранятся элементы стека.
   - `TStack()` – конструктор по умолчанию.
   - `TStack(const TStack<ValType>&)` – конструктор копирования.

- `~TStack()` – деструктор.
- `bool IsEmpty() const` – константный метод, который возвращает `true`, если стек пуст, иначе `false`.
- `bool IsFull() const` - константный метод, который возвращает `true`, если стек полон, иначе `false`.
- `void Push(ValType)` – метод, который добавляет на вершину стека переменную типа `ValType`.
- `void Pop()` – метод, который удаляет элемент на вершине стека.
- `ValType TopWatch() const` – константный метод, который только возвращает элемент на вершине стека.
- `int GetSize() const` – константный метод, который возвращает занимаемый размер стека.

4. Статический класс RPN – шаблонный класс для работы с арифметическими выражениями.

Статический класс, который содержит в себе статические методы для работы с арифметическими выражениями, для преобразования их в обратную польскую запись и соответственно для вычисления значений выражений. Класс содержит в себе:

- `static int GetPriority(const char)` – статический метод, который возвращает приоритет операции, то есть параметра метода (чем выше число, тем выше приоритет).
- `static bool IsOperation(const char)` – статический метод, который возвращает `true`, если параметр, то есть символ, является операцией, иначе `false`.
- `static void GetCountAndListOfVariables(int&, string&)` – статический метод, который на вход получает число – количество уникальных переменных и строку – выражение. В процессе работы метода параметры меняют свои значения. Количество различных переменных становится актуальных, а строка превращается в список различных имен переменных.
- `static TCouple<ValType>* SetDataOfVariables(string, int&)` – статический метод, который на вход получает строку из уникальных символов(имен переменных) и их количество. На выходе получается массив типа TCouple, то есть некоторую базу данных, содержащую в себе список имен переменных и соответствующие им значения.
- `static string CreateRPN(string)` – статический метод с, который на вход получает строку – выражение, а возвращает строку, которая является преобразованным в обратную польскую запись выражением.
- `static double CalculateRPN(string, TCouple<ValType>*, int)` – статический метод, который на вход получает строку – выражение в форме обратной польской записи, указатель на область памяти, в которой хранятся пары данных типа `TCouple`, и соответственно количество таких пар. Метод же возвращает значение выражения с данными значениями.

Описание алгоритмов

Описание структуры данных стек.

В данной программе реализована такая структура данных, как стек на базе массивов. Стек представляет собой список элементов, организованный по принципу LIFO, то есть «последним пришел, первым вышел» (Схема 1).

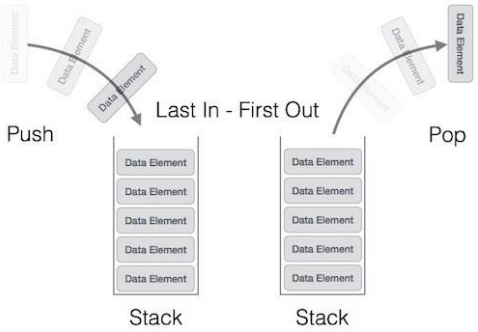

Схема 1. *Принцип работы стека.*

Количество элементов можно узнать по индексу первого свободного в стеке элементу.

Характеристики стека:

- Максимальное количество элементов, которые может в себя вместить стек (то есть некоторый настоящий его размер).
- Массив элементов.
- Индекс вершины стека, то есть индекс первого свободного элемента.

Операции над стеком:

- Положить элемент.
- Посмотреть элемент, который хранится на вершине стека.
- Удалить элемент.
- Проверка на полноту.
- Проверка на пустоту.

Описание алгоритмов преобразования в польскую запись и ее вычисления

Для упрощения описания алгоритмов введем приоритет для допустимых в выражениях арифметических операций:

- ‘*’ и ‘/’ – приоритет 3.
- ‘+’ и ‘—’ – приоритет 2.
- ‘(’ и ‘)’ – приоритет 1**.**

Преобразование выражений в обратную польскую запись.

Пусть выражение, которое вводится в консоль, хранится в переменной *str*. Для хранения операций будем использовать стек *Stack1*, а для операций *Stack2*.

1. Строка *str* просматривается слева направо до тех пор, пока не закончится.
   1.1. Если символ является пробелом, то переходим к следующему символу строки.
   1.2. Если очередной символ – символ операции, то содержимое попадает в *Stack1*.
   1.3. Если очередной символ – ‘*(*‘, то кладем в *Stack2*.
   1.4. Если выпал символ - ‘*)*’, то все операции из *Stack1* переносятся в *Stack2* до появления левой скобки. Саму левую скобку удаляем из *Stack1*, а правую никуда не помещаем.
   1.5. Если очередной символ – операция, то смотрим приоритет операции, которой хранится на вершине стека *Stack1*. Если ее приоритет выше, чем очередного символа, то перекладываем все операции из *Stack1*, у которых приоритет выше или больше текущей, в *Stack2*. Затем сам текущий символ заносим в *Stack1*.
2. Дойдя до конца выражения *str*, все оставшиеся операции в *Stack1* переносим в *Stack2*.
3. В *Stack2* будет хранится обратная польская запись.

Вычисление значения выражения в форме обратной польской записи.

Для хранения значения операндов создадим стек *Stack*, а в переменной *str* будет хранится выражение в форме обратной польской записи. Кроме того, создадим массив *TData* типа *TCouple* для хранения пары данных – имени переменной и ее значения.

1. Строка *str* снова просматривается слева направо.
   1.1. Если встретилась переменная, то находим ее имя в массиве *TData* и соответствующее ее значение, которое хранится в этом же массиве с тем же индексом, что и имя этой переменной, кладем в Stack.
   1.2. Если очередной символ – символ операции, то изымаем из Stack сначала первый элемент – *rightOperand*, а затем второй – *leftOperand*. Выполняем данную операцию в порядке названий переменных, то есть: *leftOperand <операция> rightOperand*. Если операция - деление, а правый операнд – ноль, то вызываем исключение. Само значение операции кладем обратно в *Stack*.
2. По окончании просмотра строки в *Stack* будет хранится единственная переменная – значение выражения.

Заключение

В результате данной лабораторной работы были разработаны класс, описывающий такую структуру данных, как стек; класс, благодаря которому можно работать с двумя переменными, которые связаны между собой; статический класс, позволяющий построить обратную польскую запись для какого-либо допустимого арифметического выражения с использованием стека и затем с помощью нее вычислить его значение.

Программное решение было продемонстрировано с помощью нахождения значения для введенного пользователем арифметического выражения. Описание примера работы с программой было представлено в разделе «Руководство пользователя».

Литература

1. Электронный ресурс.
   1.1. Википедия - свободная электронная энциклопедия: на русском языке [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Обратная_польская_запись.
   1.2. Алгоритмы, методы, исходники [Электронный ресурс] // URL: http://algolist.manual.ru/
2. Учебники, рабочие материалы.
   2.1. Гергель В. П. Рабочие материалы к учебному курсу «Методы программирования», часть 1 – Нижний Новгород.

Приложения

Приложение 1. Исходный код основной функции

```
#include "Includes/RPN.h"
#include "Includes/TCouple.h"

#include <string.h>
#include <iostream>

using namespace std;

void main()
{
    cout << "##### Reverse Polish notation #####" << endl << endl;
    cout << "Enter string to convert to Revers Polish notation and press <Enter>: ";

    try
    {
        string str;
        getline(cin, str);

        cout << "Converting..." << endl;
        string rpn = RPN<double>::CreateRPN(str);
        cout << "Reverse Polish Notation: " << rpn << endl;

        cout << "Enter variables and its values: " << endl;
        TCouple<double>* data = new
TCouple<double>[RPN<double>::GetCountVariables(str)];

        char* variables = RPN<double>::GetListOfVariables(str);
        for (int i = 0; i < RPN<double>::GetCountVariables(str); i++)
        {
            data[i].var = variables[i];
            cout << "Variable " << variables[i] << " : ";
            cin >> data[i].value;
        }

        double result = RPN<double>::CalculateRPN(rpn, data,
RPN<double>::GetCountVariables(str));
        cout << "Result: " << result;
        //char str[] = "A + B   * (C -D)/(F+E)+K";
    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }
}
```

Приложение 2. Класс Exceptions

```
#ifndef _EXCEPTIONS_H_
#define _EXCEPTIONS_H_

#include <iostream>
#include <string>
#include <exception>

using namespace std;

class Exception : public exception
```

```
{
private:
    string msg;
public:
    Exception(string _msg) : msg(_msg) {};

    const char* what() const noexcept
    {
        return msg.c_str();
    }
};

#endif
```

Приложение 3. Класс TCouple

```
#ifndef _COUPLE_H_
#define _COUPLE_H_

using namespace std;

template<typename ValType>
class TCouple
{
public:
    char var;
    ValType value;

    TCouple();
};

template<typename ValType>
TCouple<ValType>::TCouple()
{
    var = 0;
    value = 0;
}

#endif
```

Приложение 4. Класс TStack

```
#ifndef _TSTACK_H_
#define _TSTACK_H

#include "exceptions.h"

using namespace std;

template<typename ValType>
class TStack
{
private:
    int size;
    int top;
    ValType* elem;
public:
    TStack(int);
    TStack(const TStack<ValType>&);
    ~TStack();

    bool IsEmpty() const;
    bool IsFull() const;
```

```
    void Push(ValType);
    ValType Pop();
    ValType TopWatch();

    int GetSize() const;
};

//-----------------------------------------------------

template<typename ValType>
TStack<ValType>::TStack(int _size) : size(_size)
{
    if (size < 0)
        throw Exception("Not correct size of stack!");

    elem = new ValType[size];
    memset(elem, 0, sizeof(ValType) * size);

    top = 0;
};

template<typename ValType>
TStack<ValType>::TStack(const TStack<ValType>& _copy) : size(_copy.size),
top(_copy.top)
{
    elem = new ValType[size];
    memset(elem, _copy.elem, sizeof(ValType) * size);
};

template<typename ValType>
TStack<ValType>::~TStack()
{
    size = 0;
    top = 0;
    delete[] elem;
};

template<typename ValType>
bool TStack<ValType>::IsEmpty() const
{
    return(top == 0);
};

template<typename ValType>
bool TStack<ValType>::IsFull() const
{
    return(top == size);
};

template<typename ValType>
void TStack<ValType>::Push(ValType _object)
{
    if (IsFull())
        throw Exception("Error: stack is full!");

    elem[top++] = _object;
};

template<typename ValType>
ValType TStack<ValType>::Pop()
{
    if (IsEmpty())
        throw Exception("Error: stack is empty!");

    return (elem[--top]);
};
```

```
template<typename ValType>
ValType TStack<ValType>::TopWatch()
{
    if (IsEmpty())
        throw Exception("Error: stack is empty!");

    return (elem[top - 1]);
};

template<typename ValType>
int TStack<ValType>::GetSize() const
{
    return top;
}

#endif
```

Приложение 5. Класс RPN

```
#ifndef _RPN_H_
#define _RPN_H_

#include "TStack.h"
#include "TCouple.h"
#include "exceptions.h"

#include <cstring>
#include <cctype>

using namespace std;

template<typename ValType>
class RPN
{
private:
    static int GetPriority(const char);
    static bool IsOperation(const char);

public:
    static int GetCountVariables(string);
    static char* GetListOfVariables(string);
    static string CreateRPN(string);
    static double CalculateRPN(string, TCouple<ValType>*, int);
};

//-------------------------------------------------------------

template<typename ValType>
int RPN<ValType>::GetPriority(const char _oper)
{
    switch (_oper)
    {
    case '(':
        return 1;
    case ')':
        return 1;
    case '+':
        return 2;
    case '-':
        return 2;
    case '*':
        return 3;
    case '/':
        return 3;
    default:
```

```
      throw Exception("Error: Invalid character!");
    }
};

template<typename ValType>
bool RPN<ValType>::IsOperation(const char _s)
{
    return (_s == '+' || _s == '-' || _s == '*' || _s == '/' ||
        _s == '(' || _s == ')');
};

template<typename ValType>
int RPN<ValType>::GetCountVariables(string _str)
{
    if (_str.length() == 0)
        return 0;

    int count = 0;
    char* vars = new char[_str.length() + 1];
    memset(vars, 0, sizeof(char) * (_str.length() + 1));

    for (int i = 0; i < _str.length(); i++)
    {
        char symbol = static_cast<char>(_str[i]);

        if ((symbol != ' ') && (!IsOperation(symbol)) && (strchr(vars, symbol) ==
NULL))
                vars[count++] += symbol;
    }

    return count;
};

template<typename ValType>
char* RPN<ValType>::GetListOfVariables(string _str)
{
    int count = 0;
    char* vars = new char[GetCountVariables(_str) + 1];
    memset(vars, 0, sizeof(char) * (GetCountVariables(_str) + 1));

    for (int i = 0; i < _str.length(); i++)
    {
        char symbol = static_cast<char>(_str[i]);

        if ((symbol != ' ') && (!IsOperation(symbol)) && (strchr(vars, symbol) ==
NULL))
            vars[count++] += symbol;
    }

    return vars;
}

template<typename ValType>
string RPN<ValType>::CreateRPN(string _str)
{
    if (_str.length() == 0)
        throw Exception("Error: String is empty!");

    TStack<char> st1(_str.length() + 1);  // operations
    TStack<char> st2(_str.length() + 1);  // operands

    for (int i = 0; i < _str.length(); i++)
    {
        char symbol = static_cast<char>(_str[i]);
```

```
        if (symbol == ' ')  // space processing
            continue;

        if (IsOperation(symbol))  // operation processing
        {
            if (symbol == '(')
            {
                st1.Push(symbol);
                continue;
            }

            if (symbol == ')')
            {
                while (st1.TopWatch() != '(')
                    st2.Push(st1.Pop());

                st1.Pop(); // delete '('
                continue;
            }

            if ((st1.IsEmpty()) || (GetPriority(symbol) >=
GetPriority(st1.TopWatch())))
            {
                st1.Push(symbol);
                continue;
            }

            while((!st1.IsEmpty()) && (GetPriority(symbol) <=
GetPriority(st1.TopWatch())))
                st2.Push(st1.Pop());

            st1.Push(symbol);
            continue;
        }

        if (isalpha(symbol))  // variable processing
        {
            st2.Push(symbol);
            continue;
        }

        throw Exception("Error: The symbols are not correct in string!");  /* no
operation, no space, no letter*/
    }

    while (!st1.IsEmpty())
    {
        char tmp = st1.Pop();
        st2.Push(tmp);
    }

    string rpn(st2.GetSize(), 0);

    while (!st2.IsEmpty())
    {
        rpn[st2.GetSize() - 1] = st2.TopWatch();
        st2.Pop();
    }

    return rpn;
};

template<typename ValType>
double RPN<ValType>::CalculateRPN(string _str, TCouple<ValType>* _data, int _
countData)
{
```

```
    if (_str.length() == 0)
        throw Exception("Error: String is empty!");


    TStack<double> value(_str.length());

    for (int i = 0; i < _str.length(); i++)
    {
        char symbol = static_cast<char>(_str[i]);

        if (!IsOperation(symbol))
        {
            for (int j = 0; j < _countData; j++)
            {
                if (_data[j].var == symbol)
                {
                    value.Push(static_cast<double>(_data[j].value));
                    break;
                }
            }

            continue;
        }

        double rightOperand = value.Pop();
        double leftOperand = value.Pop();
        double res = 0;

        switch (symbol)
        {
        case '+':
            res = leftOperand + rightOperand;
            break;
        case '-':
            res = leftOperand - rightOperand;
            break;
        case '*':
            res = leftOperand * rightOperand;
            break;
        case '/':
            if (rightOperand == 0)
                throw Exception("Error: Division by zero!");
            res = leftOperand / rightOperand;
            break;
        }

        value.Push(res);
    }

    return value.Pop();
};

#endif
```