**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по курсовой работе**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Преобразование алгебраических формул из инфиксной в постфиксную форму записи и вычисление значения выражения»**

**Вариант 19(1)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

# Постановка задачи

Преобразование алгебраических формул из инфиксной в постфиксную форму записи и вычисление значения выражения.

**Обоснование выбора используемых структур данных**

Postfix – класс постфикса и преобразований в нем

Node – класс Node

List – класс списка введенной строки

Stack – класс стека

Node\_S – вложенный класс стека

Node\* next = nullptr; - указатель на следующий элемент.

string data; - данные в строке.

size\_t size; - размер.

int first\_action = -1; - вспомогательная переменная для того, чтобы понять, какие действия вычислять в первую очередь.

Node\* head; - указатель на первый элемент.

Node\* last; - указатель на последний элемент.

Node\* elem; - указатель на элемент списка.

Node\_Q\* next; - указатель на следующий элемент стека.

Node\_S\* first\_S; - указатель на первый элемент стека.

Node\_S\* last\_S; - указатель на последний элемент стека.

Stack\* Stack\_elem; - стек (где временно хранятся элементы).

List\* List\_postfix; - выгрузка измененного выражения.

bool List::IsEmpty()- проверка на пустоту.

void List::push\_back(Node\* new\_Elem) - добавление нового элемента.

void List::push\_back\_string(string new\_Elem, int first\_action) – добавление элемента в стек, если строка.

void List::print\_to\_console() – вывод в консоль.

void List::clear() – очистка List.

bool Stack::IsEmpty() - проверка на пустоту стека.

void Stack::push(Node\* new\_elem) - добавление элемента в стек.

void Stack::push\_string(string data, int first\_action) – добавление элемента в стек, если строка.

void Stack::push\_char(char new\_elem, int first\_action) – добавление элемента в стек, если элемент char.

Node\* Stack::pop() – удаление первого элемента стека.

Node\* Stack::front()- возвращение первого элемента стека.

bool Postfix::Number(char Symbol) – проверка введенного символа (число или нет).

bool Postfix::Function(char Symbol) – проверка введенного символа (функция или нет).

int Postfix::WhichOperator(char Symbol) – проверка, какой оператор введен.

double Postfix::Operator\_last(char Symbol, double A1, double A2) – результат после оператора.

int Postfix::WhichFunction(string function) – проверка, какая функция введена.

double Postfix::Function\_last(int Function, double Argument) – результат после функции.

void Postfix::Infix\_to\_Postfix(string read) – преобразование инфиксной формы в постфиксную.

double Postfix::process() - процесс вычислений.

void Postfix::out()- функция вывода.

int main() – функция для ввода и вывода результата (исполняемая)

# Описание алгоритма решения

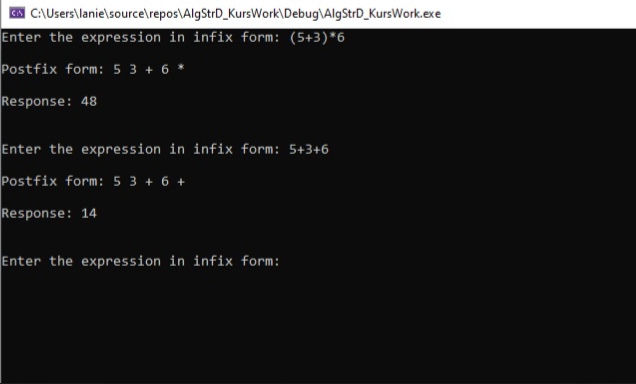
Программа последовательно проходит по всем введенным символам. Так, если встречается число, оно сразу выводится в итоговую строку. Если символ – открывающая (левая) скобка, он помещается в стек. Если символ – закрывающая (правая) скобка, то происходит выгрузка символов из стека, пока не встретится открывающая (левая) скобка. При этом обе скобки удаляются. Если символ – символ оператора, тогда:

а) Если стек пуст, если элемент в вершине имеет меньший приоритет, чем приоритет текущего элемента, то переносим текущий элемент в стек.

б) Если элемент на вершине стека имеет приоритет, больший или равный приоритету текущего элемента, то происходит выгрузка символов из стека до тех пор, пока выполняется это условие.

Далее вычисляем значение выражений.

# Пример работы



# Листинг

**Node.h**

#pragma once

#include <iostream>

using namespace std;

class Node {

public:

Node\* next = nullptr;

string data;

size\_t size;

int first\_action = -1;

Node(string data, int first\_action = -1, Node\* next = nullptr) {

this->data = data;

this->first\_action = first\_action;

this->next = next;

};

~Node() {};

};

**List.h**

#pragma once

#include <iostream>

#include <string>

#include "Node.h"

using namespace std;

class List

{

private:

Node\* head;

Node\* last;

public:

List();

bool IsEmpty();

Node\* set\_head() {

return head;

};

void push\_back(Node\* new\_Elem);

void push\_back\_string(string new\_Elem, int first\_action);

void print\_to\_console();

void clear();

~List();

};

**Stack.h**

#pragma once

#include "List.h"

class Stack {

private:

class Node\_S {

public:

Node\_S(Node\* elem = nullptr, Node\_S\* next = nullptr) {

this->elem = elem;

this->next = next;

};

Node\_S(string data = "\_\_", int first\_action = -1, Node\_S\* next = nullptr) {

this->elem = new Node(data, first\_action, nullptr);

this->next = next;

};

~Node\_S() {};

Node\* elem;

Node\_S\* next;

};

Node\_S\* first\_S;

Node\_S\* last\_S;

public:

Stack();

bool IsEmpty();

void push(Node\*);

void push\_string(string data, int first\_action);

void push\_char(char new\_elem, int first\_action);

Node\* pop();

Node\* front();

~Stack();

};

**Postfix.h**

#pragma once

#include "Stack.h"

#include <stdexcept>

#include <cmath>

#include <string>

class Postfix {

private:

Stack\* Stack\_elem;

List\* List\_postfix;

enum first\_action {

left\_brecket, right\_brecket, add\_sub, multipl\_division, degree, func, not\_operator = -1

};

enum functions {

Sin, Cos, Log, Ln, Tg, Ctg, Sqrt, Abs, Exp, Pi ,E

};

public:

Postfix();

bool Number(char Symbol);

bool Function(char Symbol);

int WhichOperator(char Symbol);

double Operator\_last(char Symbol, double A1, double A2);

int WhichFunction(string Function);

double Function\_last(int Function, double Argument);

void Infix\_to\_Postfix(string read);

double process();

void out();

~Postfix();

};

**List.cpp**

#include "List.h"

List::List(){

head = nullptr;

last = nullptr;

}

bool List::IsEmpty() {

return (head == nullptr);

}

void List::push\_back(Node\* new\_Elem)

{

if (!(IsEmpty())) {

last->next = new\_Elem;

last = last->next;

}

else {

head = new\_Elem;

head->next = nullptr;

last = head;

}

}

void List::push\_back\_string(string new\_Elem, int first\_action)

{

if (!(IsEmpty())) {

last->next = new Node(new\_Elem, first\_action);

last = last->next;

}

else {

head = new Node(new\_Elem, first\_action);

last = head;

}

}

void List::print\_to\_console() {

Node\* cur = head;

while (cur != nullptr) {

cout << cur->data << ' ';

cur = cur->next;

}

cout << endl;

cout << endl;

}

void List::clear() {

Node\* cur = head;

while (!(this->IsEmpty())) {

cur = head;

head = cur->next;

delete cur;

}

if (IsEmpty()) {

throw out\_of\_range("List is empty");

}

}

List::~List() {

clear();

}

**Stack.cpp**

#include "Stack.h"

#include <stdexcept>

Stack::Stack() {

first\_S = nullptr;

last\_S = nullptr;

}

bool Stack::IsEmpty() {

return (first\_S == nullptr);

}

void Stack::push(Node\* new\_elem) {

first\_S = new Node\_S(new\_elem, first\_S);

}

void Stack::push\_string(string data, int first\_action) {

first\_S = new Node\_S(data, first\_action, first\_S);

}

void Stack::push\_char(char new\_elem, int first\_action) {

string insert;

insert.push\_back(new\_elem);

first\_S = new Node\_S(insert, first\_action, first\_S);

}

Node\* Stack::pop() {

Node\* a = first\_S->elem;

first\_S = first\_S->next;

return a;

}

Node\* Stack::front() {

return first\_S->elem;

}

Stack::~Stack() {

while (!IsEmpty()) {

pop();

}

}

**Postfix.cpp**

#include "Postfix.h"

double pi = 3.14159265358979323846;

double e = 2.71828182845904523536;

Postfix::Postfix() {

Stack\_elem = new Stack();

List\_postfix = new List();

}

bool Postfix::Number(char Symbol) {

int i = 0;

string Numbers = {

"1234567890.,"

};

for (char CurNumber : Numbers) {

if (CurNumber == Symbol) {

return true;

}

}

return false;

}

bool Postfix::Function(char Symbol) {

int i = 0;

string Symbols = {

"abcdefghijklmnopqrstuvwxyz"

};

for (char CurSymbol : Symbols) {

if (CurSymbol == Symbol) {

return true;

}

}

return false;

}

int Postfix::WhichOperator(char Symbol) {

if (Symbol == '(') {

return left\_brecket;

}

else {

if (Symbol == ')') {

return right\_brecket;

}

else {

if (Symbol == '+' || Symbol == '-') {

return add\_sub;

}

else {

if (Symbol == '\*' || Symbol == '/') {

return multipl\_division;

}

else {

if (Symbol == '(') {

return left\_brecket;

}

else {

if (Symbol == ')') {

return right\_brecket;

}

else {

if (Symbol == '^') {

return degree;

}

else {

return not\_operator;

}

}

}

}

}

}

}

}

double Postfix::Operator\_last(char Symbol, double A1, double A2) {

double Response = 0;

switch (Symbol) {

case '+': {

Response = A1 + A2;

break;

}

case '-': {

Response = A1 - A2;

break;

}

case '\*': {

Response = A1 \* A2;

break;

}

case '/': {

Response = A1 / A2;

break;

}

case '^': {

Response = pow(A1, A2);

break;

}

}

return Response;

}

int Postfix::WhichFunction(string function) {

if (function == "cos") {

return Cos;

}

else {

if (function == "sin") {

return Sin;

}

else {

if (function == "tg") {

return Tg;

}

else {

if (function == "ctg") {

return Ctg;

}

else {

if (function == "log") {

return Log;

}

else {

if (function == "ln") {

return Ln;

}

else {

if (function == "sqrt") {

return Sqrt;

}

else {

if (function == "abs") {

return Abs;

}

else {

if (function == "exp") {

return Exp;

}

else {

if (function == "pi") {

return Pi;

}

else {

if (function == "e") {

return E;

}

else {

return not\_operator;

}

}

}

}

}

}

}

}

}

}

}

}

double Postfix::Function\_last(int Function, double Argument) {

double Response = 0;

switch (Function) {

case Cos: {

Response = cos(Argument);

break;

}

case Sin: {

Response = sin(Argument);

break;

}

case Tg: {

Response = tan(Argument);

break;

}

case Ctg: {

Response = 1 / (tan(Argument));

break;

}

case Log: {

Response = log10(Argument);

break;

}

case Ln: {

Response = log(Argument);

break;

}

case Sqrt: {

Response = sqrt(Argument);

break;

}

case Abs: {

Response = abs(Argument);

break;

}

case Exp: {

Response = exp(Argument);

break;

}

}

return Response;

}

void Postfix::Infix\_to\_Postfix(string read) {

size\_t i = 0, left\_breckets = 0, right\_breckets = 0, dots = 0;

string Cur\_postfix;

/\*for (i = 0; i < read.size(); i++)\*/

while (i < read.size()) {

Cur\_postfix.clear();

if (read[i] == '-' && List\_postfix->IsEmpty()) {

Cur\_postfix.push\_back('0');

List\_postfix->push\_back\_string(Cur\_postfix, not\_operator);

Cur\_postfix.clear();

Stack\_elem->push\_char(read[i++], add\_sub);

}

if (read[i] == ' ') {

i++;

}

if (Number(read[i])) {

dots = 0;

while (Number(read[i])) {

if (read[i] == '.') {

dots++;

}

Cur\_postfix.push\_back(read[i++]);

}

if (read[i] == ' ') {

i++;

}

if (read[i] == '(') {

throw ("Entry error, please try again");

}

List\_postfix->push\_back\_string(Cur\_postfix, not\_operator);

Cur\_postfix.clear();

}

if (WhichOperator(read[i]) == left\_brecket) {

left\_breckets++;

Stack\_elem->push\_char(read[i++], left\_brecket);

if (read[i] == '-') {

Cur\_postfix.push\_back('0');

List\_postfix->push\_back\_string(Cur\_postfix, not\_operator);

Cur\_postfix.clear();

Stack\_elem->push\_char(read[i++], add\_sub);

}

if (WhichOperator(read[i]) == right\_brecket) {

throw ("Entry error, please try again");

}

}

else {

if (WhichOperator(read[i]) == right\_brecket) {

right\_breckets++;

while ((!Stack\_elem->IsEmpty()) && (Stack\_elem->front()->first\_action != left\_brecket)) {

List\_postfix->push\_back(Stack\_elem->pop());

}

Stack\_elem->pop();

i++;

}

else {

if (WhichOperator(read[i]) != not\_operator) {

if (Stack\_elem->IsEmpty()) {

Stack\_elem->push\_char(read[i], WhichOperator(read[i]));

}

else {

if (!Stack\_elem->IsEmpty()) {

if (Stack\_elem->front()->first\_action < WhichOperator(read[i])) {

Stack\_elem->push\_char(read[i], WhichOperator(read[i]));

}

}

else {

if (!Stack\_elem->IsEmpty()) {

if (Stack\_elem->front()->first\_action >= WhichOperator(read[i])) {

while ((!Stack\_elem->IsEmpty()) && (Stack\_elem->front()->first\_action >= WhichOperator(read[i]))) {

List\_postfix->push\_back(Stack\_elem->pop());

}

Stack\_elem->push\_char(read[i], WhichOperator(read[i]));

}

}

}

}

i++;

}

if (Function(read[i])) {

string function;

while (Function(read[i]))

function.push\_back(read[i++]);

if (WhichFunction(function) != not\_operator) {

if (Stack\_elem->IsEmpty()) {

Stack\_elem->push\_string(function, func);

}

else {

if (!Stack\_elem->IsEmpty()) {

if (Stack\_elem->front()->first\_action < func) {

Stack\_elem->push\_string(function, func);

}

}

else {

if (!Stack\_elem->IsEmpty()) {

if (Stack\_elem->front()->first\_action >= func) {

while (!Stack\_elem->IsEmpty() && (Stack\_elem->front()->first\_action >= func)) {

List\_postfix->push\_back(Stack\_elem->pop());

}

Stack\_elem->push\_string(function, func);

}

}

}

}

if (read[i] == ' ') {

i++;

}

if (WhichOperator(read[i]) == left\_brecket) {

left\_breckets++;

Stack\_elem->push\_char(read[i++], left\_brecket);

if (read[i] == '-') {

Cur\_postfix.push\_back('0');

List\_postfix->push\_back\_string(Cur\_postfix, not\_operator);

Cur\_postfix.clear();

Stack\_elem->push\_char(read[i++], add\_sub);

}

if (Number(read[i])) {

dots = 0;

while (Number(read[i])) {

if (read[i] == '.') {

dots++;

}

Cur\_postfix.push\_back(read[i++]);

}

if (read[i] == ' ') {

i++;

}

if (dots > 1) {

throw ("Entry error, please try again");

}

if (read[i] == '(') {

throw ("Entry error, please try again");

}

List\_postfix->push\_back\_string(Cur\_postfix, not\_operator);

Cur\_postfix.clear();

}

}

if (WhichOperator(read[i]) == right\_brecket) {

right\_breckets++;

while (Stack\_elem->front()->first\_action != left\_brecket) {

List\_postfix->push\_back(Stack\_elem->pop());

}

Stack\_elem->pop();

i++;

}

}

}

}

}

if (read[i] == ' ') {

i++;

}

if (i == read.size()) {

break;

}

if (!Number(read[i]) && !Function(read[i]) && WhichOperator(read[i]) == not\_operator) {

throw ("Entry error, please try again");

break;

}

}

if (!Stack\_elem->IsEmpty()) {

while (!Stack\_elem->IsEmpty()) {

List\_postfix->push\_back(Stack\_elem->pop());

}

}

}

double Postfix::process() {

Node\* Cur = List\_postfix->set\_head();

double A1 = 0, A2 = 0, Response = 0;

string number;

number.clear();

Stack\_elem = new Stack();

while (Cur != nullptr) {

if (Number(Cur->data[0])) {

Stack\_elem->push\_string(Cur->data, not\_operator);

}

if (WhichFunction(Cur->data) == Pi) {

Stack\_elem->push(Cur);

}

if (WhichFunction(Cur->data) == E) {

Stack\_elem->push(Cur);

}

if (WhichOperator(Cur->data[0]) != not\_operator) {

number.clear();

if (!Stack\_elem->IsEmpty()) {

number = Stack\_elem->pop()->data;

if (WhichFunction(number) == Pi) A2 = pi;

else if (WhichFunction(number) == E) A2 = e;

else A2 = stod(number);

number.clear();

}

else throw("Entry error, please try again");

if (!Stack\_elem->IsEmpty()) {

number = Stack\_elem->pop()->data;

if (WhichFunction(number) == Pi) A1 = pi;

else if (WhichFunction(number) == E) A1 = e;

else A1 = stod(number);

number.clear();

}

else throw("Entry error, please try again");

Response = Operator\_last(Cur->data[0], A1, A2);

Stack\_elem->push\_string(to\_string(Response), not\_operator);

}

if (WhichFunction(Cur->data) != not\_operator) {

if (WhichFunction(Cur->data) != Pi) {

if (WhichFunction(Cur->data) != E) {

number.clear();

if (!Stack\_elem->IsEmpty()) {

number = Stack\_elem->pop()->data;

if (WhichFunction(number) == Pi) {

A1 = pi;

}

else {

if (WhichFunction(number) == E) {

A1 = e;

}

else {

A1 = stod(number);

}

number.clear();

}

}

else {

throw("Entry error, please try again");

}

Response = Function\_last(WhichFunction(Cur->data), A1);

Stack\_elem->push\_string(to\_string(Response), not\_operator);

}

}

}

Cur = Cur->next;

}

if (!Stack\_elem->IsEmpty()) {

if (Response == 0) {

Response = stod(Stack\_elem->pop()->data);

}

}

return Response;

}

void Postfix::out() {

List\_postfix->print\_to\_console();

}

Postfix::~Postfix() {

List\_postfix->~List();

Stack\_elem->~Stack();

}

int main() {

Postfix\* Postfix\_elem = nullptr;

double Response;

string read;

while (1) {

Postfix\_elem = new Postfix();

cout << "Enter the expression in infix form: ";

getline(cin, read);

cout << endl;

if (read == "Exit") {

break;

}

try {

Postfix\_elem->Infix\_to\_Postfix(read);

cout << "Postfix form: ";

Postfix\_elem->out();

Response = Postfix\_elem->process();

cout << "Response: " << Response << endl;

cout << endl;

cout << endl;

}

catch (const char\* error) {

cout << error << endl;

}

}

return 0;

}

# Вывод

Я освоила все знания, полученные за нынешний семестр, программа успешно преобразовывает введенную инфиксную форму в постфиксную и вычисляет результат.