Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 382003-1

Смолин К.А.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Требуется написать программу, производящую вычисление арифметических выражений c вещественными числами. В рамках задания нужно разработать класс **stack**, реализующий стек, написать программу, позволяющую переводить выражение в постфиксную запись и вычислять значение выражения по польской записи, провести проверку через Google Tests. Выражение должно разбиваться на лексемы, проверяться на корректность, причём в случае обнаружения ошибок пользователю должны выводиться виды ошибок и номера символов строки, где найдены ошибки. Должна иметься возможность работы с выражениями, содержащими переменные. Также нужно реализовать пользовательский интерфейс.

# Метод решения

Одной из задач является разработка своего стека. **stack** – динамическая однородная структура данных, обеспечивающая хранение элементов и доступ к ним по принципу FILO (first in – last out). Данная структура данных должна обеспечивать как минимум добавление/удаление элементов и доступ к «верхнему» элементу (последнему добавленному в стек).

**Преобразование в обратную польскую запись:**

Алгоритм выведет строку в постфиксном порядке. Мы обрабатываем инфиксное выражение слева направо. Для начала – удаляем все пробелы из введенного выражения.

**std::string expression – стек, в который происходит запись результата.**

**stack<char> \_operators\_stack – стек, в который помещаются операторы.**

Дальше для каждого токена может возникнуть четыре случая:

1. Если текущий токен представляет собой открывающую скобку, '(', помещаем его в стек.
2. Если текущий токен является закрывающей скобкой, ')', извлекаем токены из стека до тех пор, пока не будет удалена соответствующая открывающая скобка ‘(‘. Каждый оператор добавляем в конец постфиксного выражения.
3. Если текущий токен является операндом, добавляем его в конец постфиксного выражения.
4. Если текущий токен является оператором, помещаем его в stack top. Прежде чем сделать это, сначала извлекаем содержимое стека, пока у нас не появится оператор с более низким приоритетом сверху, или стек не опустеет. Добавляем каждый оператор в конец постфиксного выражения.
5. Если текущий токен является унарным минусом, помещаем в стек ‘\_‘

**Проверка выражения:**

Проверка ошибок использует следующий набор правил:

1. Перед числом не может закрывающая скобка;
2. Если операции нет среди допустимых, это недопустимый символ;
3. Перед бинарной операцией не может идти операция или открывающая скобка;
4. Перед открывающей скобкой не может идти число или закрывающая скобка;
5. Перед закрывающей скобкой не может идти операция или открывающая скобка
6. Скобки во всём выражении образуют правильную скобочную последовательность.
7. Выражение не может оканчиваться операцией или открывающей скобкой
8. Если в выражении отсутствуют операторы, то это тоже считается ошибкой

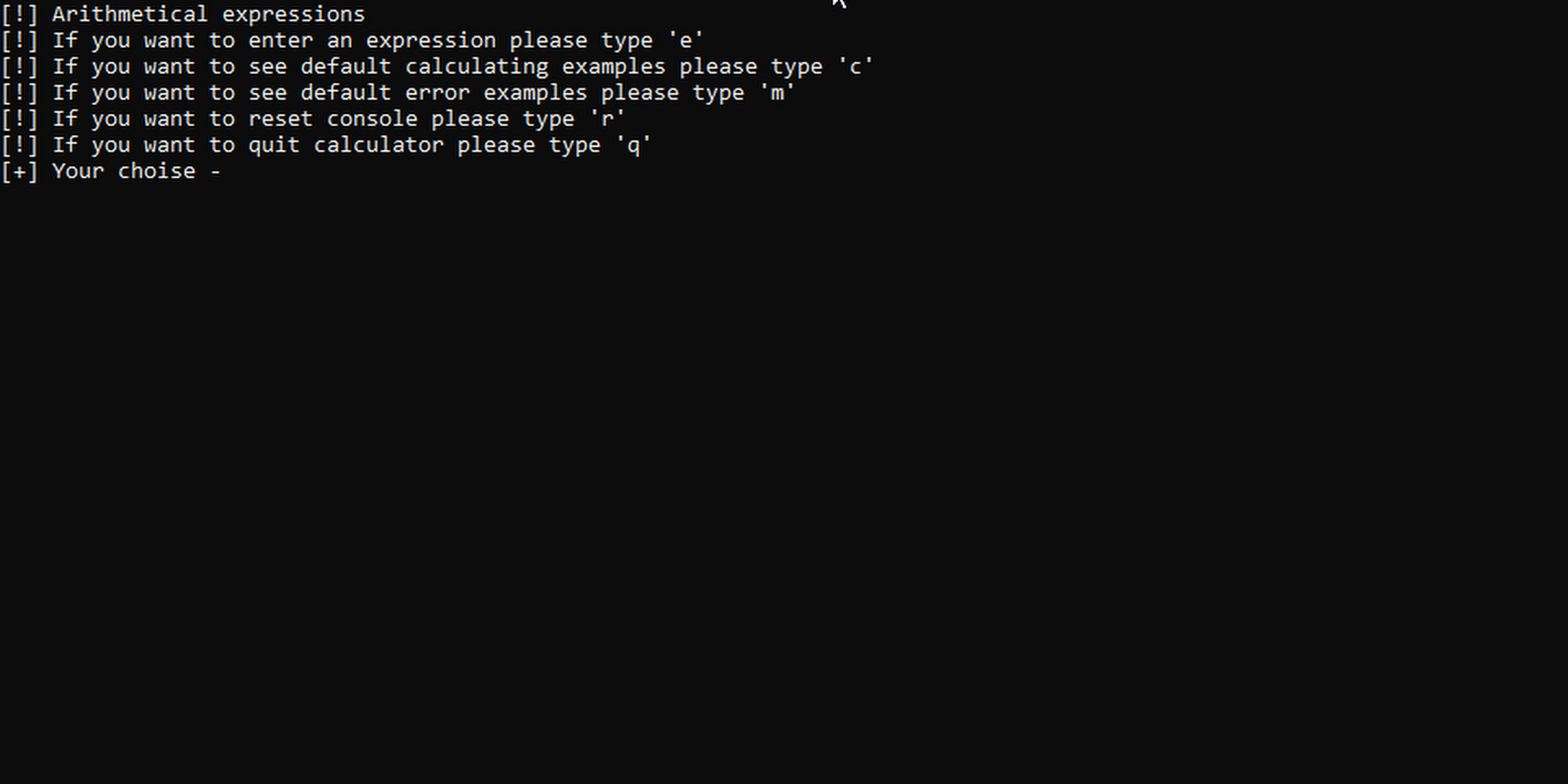
**Вычисление выражения:**

В методе **compute** сканируется постфиксное выражение.

1. Если токен – число, его значение кладётся в стек(**stack<double> operands\_stack**);
2. Если токен – унарная операция, она применяется к верхнему значению в стеке;
3. Если токен – бинарная операция, из стека извлекаются 2 верхних операнда, к ним применяется эта операция, результат кладётся в стек.

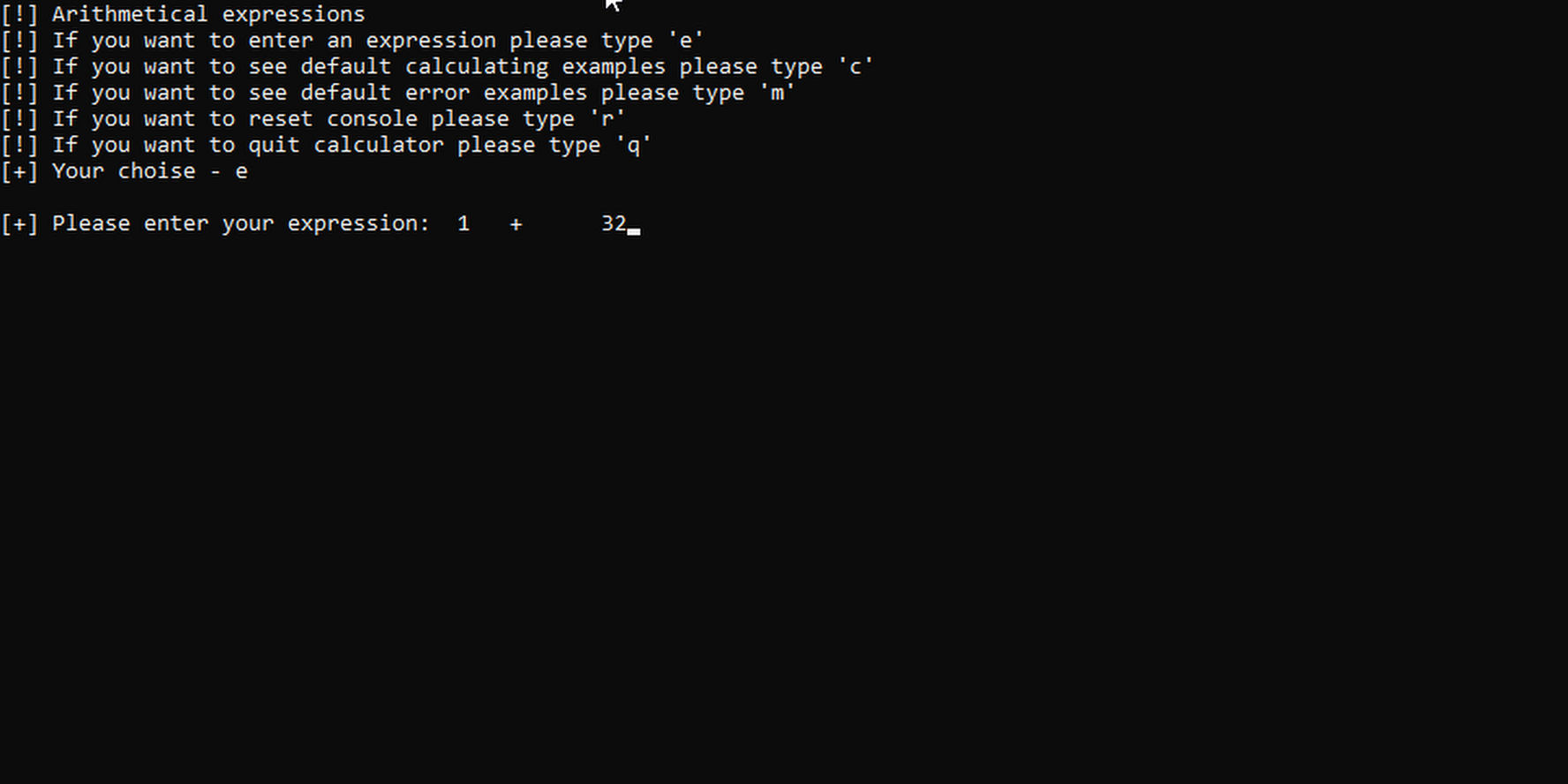
# Руководство пользователя

При запуске программы пользователя встречает следующее меню:

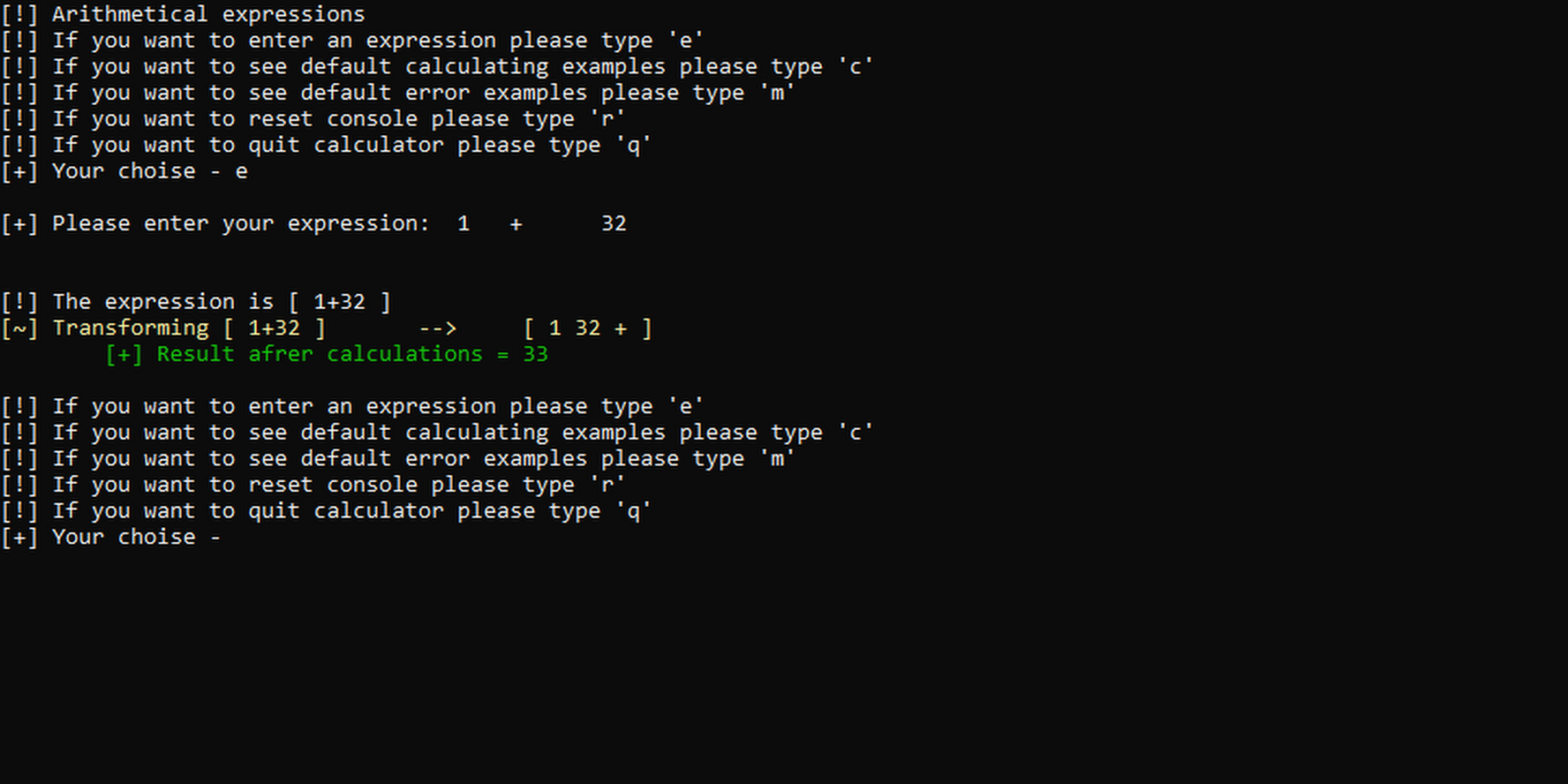


1. Если пользователь хочет работать с собственным выражением, следует ввести ‘e’
2. Существует возможность показать стандартные примеры перевода выражения в польскую запись, а также вычисления значения по польской записи. Для этого необходимо ввести ‘c’
3. Существует возможность показать стандартные примеры ошибок. Для этого необходимо ввести ‘m’
4. Для очистки окна консоли нужно ввести ‘r’
5. Для выхода можно ввести ‘q’

После ввода ‘e’ будет предложено ввести выражение. Допускаются пробелы.



После ввода и нажатия на enter, будет выведен результат, а также будет возможность выбора 5 вариантов дальнейшей работы, описанных выше.



Если пользователь ввел ‘с’, будут показаны стандартные примеры вычисления:



Если пользователь ввел ‘m’, будут показаны стандартные примеры ошибок:



Если пользователь ввел недопустимый символ при навигации в меню, консоль закроется.

# Описание программной реализации

В решение входят несколько проектов:

1. **arithmetic** – содержит классы **stack** и **arithmetic** и их реализации.

**stack.h** – реализация класса **stack**.

**arithmetic.h** – объявление класса **arithmetic**, хранящего выражение.

**arithmetic.cpp** – реализация методов класса **arithmetic**.

1. **sample** – содержит реализацию пользовательского приложения для вычисления арифметических выражений

**main-arithmetic.cpp** – содержит инициализацию меню.

**interface.cpp** – содержит реализацию меню.

1. **tests** – проект с реализацией тестов.

**stack:**

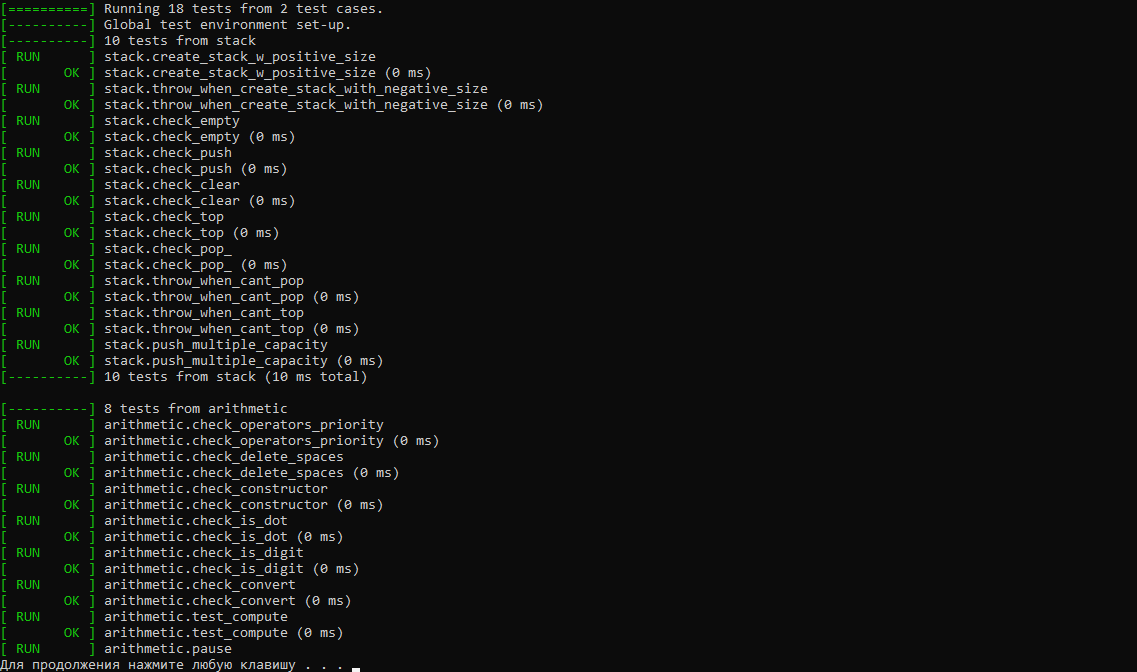
* **T\* data** – массив элементов типа T
* **size\_t capacity -** ёмкость стека(максимальное количество элементов, которое можно добавить без перевыделения памяти);
* **size\_t size -** текущее количество элементов в стеке**.**
* **stack(const size\_t cap = 0)** – конструктор с параметром, принимающий ёмкость стека.
* **stack(const stack& copy)** - конструктор копирования.
* **~stack() –** деструктор.
* **void clear()** – удаляет все элементы стека.
* **void push(const T& value)** - добавляет элемент в стек. Если **capacity** равно 0, выделяет память на 1 элемент. Если стек заполнен, производит перевыделение памяти, удваивая ёмкость стека.
* **bool empty()** – проверка на пустоту.
* **T pop()** – Удаляет верхний элемент стека.
* **T& top()** – Возвращает верхний элемент стека.

**Arithmetic:**

* **arithmetic(const std::string& \_str) : expr(\_str), copy\_expr(\_str) –** конструктор.
* **HANDLE hConsole** - Хэндл консоли, необходим для изменения цвета выводимого текста
* **delete\_spaces() –** удаляет все пробелы в введенном выражении.
* **convert() –** производит преобразование в обратную польскую запись.
* **compute(const std::string&) –** вычисляет значение по польской записи.
* **is\_expression\_is\_correct(const std::string&) –** проверка выражения.
* **letters\_counter(const std::string&) –** счетчик букв.
* **operators\_priority(char operators) –** приоритет операторов.
* **is\_digit(char elem) –** проверка, является ли элемент цифрой.
* **is\_letter(char elem) -** проверка, является ли элемент буквой.
* **is\_operator(char op) -** проверка, является ли элемент оператором.
* **is\_dot(char elem)** - проверка, является ли элемент точкой**.**
* **void print() –** вывод результата.

# Подтверждение корректности

С целью проверки правильности работы модули проходили тестирование с помощью библиотеки Google Tests. Проверялись классы **stack** и **arithmetic**. Потребовалось написать свои тесты для покрытия большинства возможных случаев использования. Результаты прохождения тестов следующие:



# Заключение

Поставленная задача выполнена. Реализованы шаблонный стек и класс для вычисления арифметических выражений; создано корректно работающее при различных вводимых данных пользовательское приложение.

# Приложение

std::string arithmetic::delete\_spaces() {

expr.erase(std::remove\_if(expr.begin(), expr.end(), std::isspace), expr.end());

return expr;

}

std::vector<std::string> arithmetic::is\_expression\_is\_correct(const std::string& \_str) {

size\_t c\_brackets = 0;

size\_t op\_brackets = 0;

size\_t operators\_counter = 0;

std::vector<std::string> errors;

SetConsoleTextAttribute(hConsole, 4);

for (size\_t i = 0; i < \_str.length(); ++i) {

if (!(is\_dot(\_str[i]) || is\_digit(\_str[i]) ||

is\_operator(\_str[i]) || is\_letter(\_str[i]) || \_str[i] == ' '

|| \_str[i] == '(' || \_str[i] == ')')) {

errors.push\_back("[-] Incorrect symbol in " + std::to\_string(i) + " position \n");

}

if (is\_operator(\_str[i])) {

operators\_counter++;

}

if (\_str[0] == '+' || \_str[0] == '\*' || \_str[0] == '/' || \_str[0] == '.') {

errors.push\_back("[-] Expression cant start with folowing operators/dot: [+],[\*],[/],[.]\n");

}

if (\_str[0] == ')') {

errors.push\_back("[-] Expression cant start with closing bracket\n");

}

if (!(is\_digit(\_str[0]) || is\_letter(\_str[0]) || \_str[0] == '(' || \_str[0] == '-')) {

errors.push\_back("[-] Expression can start only with digits or opening bracket or unary minus\n");

}

if (is\_operator(\_str[i])) {

if (\_str[i + 1] == '+' || \_str[i + 1] == '\*' || \_str[i + 1] == '/') {

errors.push\_back("[-] There are should not be an operator afrer operator: invalid position - " + std::to\_string(i + 2) + "\n");

}

else if (is\_dot(\_str[i + 1])) {

errors.push\_back("[-] There are should not be a dot afrer operator: invalid position - " + std::to\_string(i + 2) + "\n");

}

else if (is\_dot(\_str[i - 1])) {

errors.push\_back("[-] There are should not be a dot before operator: invalid position - " + std::to\_string(i) + "\n");

}

continue;

}

if (\_str[i] == ')' && \_str[i - 1] == '(') {

errors.push\_back("[-] There are should not be an opening bracket before closing bracket: invalid position - " + std::to\_string(i) + "\n");

}

if (is\_dot(\_str[\_str.length() - 1])) {

errors.push\_back("[-] Expression shoud be ended with a digit or a closing bracket\n");

}

if (is\_operator(\_str[\_str.length() - 1])) {

errors.push\_back("[-] Expect a number after the last operator: invalid position - " + std::to\_string(\_str.length()) + "\n");

}

if (\_str[\_str.length() - 1] == '(') {

errors.push\_back("[-] Expect a number after the opening bracket: invalid position - " + std::to\_string(\_str.length()) + "\n");

}

if (\_str[i] == '(' && (\_str[i + 1] == '+' || \_str[i + 1] == '/')) {

errors.push\_back(" [-] Didn't expect '+' or '/' after opening bracket \n");

}

if (\_str[i] == '(') {

op\_brackets++;

}

if (\_str[i] == ')') {

c\_brackets++;

}

if (is\_digit(\_str[i]) && \_str[i + 1] == '(') {

errors.push\_back("[-] Opening bracket after a digit were not expected\n");

}

}

if (op\_brackets > c\_brackets) {

errors.push\_back("[-] Missing closing bracket\n");

}

else if (op\_brackets < c\_brackets) {

errors.push\_back("[-] Missing opening bracket\n");

}

if (operators\_counter == 0) {

errors.push\_back("[-] There are no operators in your expression \n");

}

return errors;

}

std::string arithmetic::convert() {

/\* std::string expression is an output(operands) stack \*/

/\* stack<char> \_operators\_stack is an operators stack \*/

stack<char> \_operators\_stack;

std::string expression;

std::string entered\_string = delete\_spaces();

for (size\_t i = 0; i < entered\_string.length(); ++i) {

if (entered\_string[i] == '(')

\_operators\_stack.push(entered\_string[i]);

else if (entered\_string[i] == ')') {

while (\_operators\_stack.top() != '(') {

expression.push\_back(\_operators\_stack.top());

expression.push\_back(' ');

\_operators\_stack.pop();

}

\_operators\_stack.pop();

}

else if (is\_digit(entered\_string[i]) || is\_letter(entered\_string[i]) || is\_dot(entered\_string[i])) {

for (; i < entered\_string.length() && (is\_digit(entered\_string[i]) || is\_letter(entered\_string[i]) || is\_dot(entered\_string[i])); ++i)

expression.push\_back(entered\_string[i]);

--i;

expression.push\_back(' ');

}

else if (entered\_string[i] == '-' && (i == 0 || entered\_string[i - 1] == '(' || is\_operator(entered\_string[i - 1])))

\_operators\_stack.push('\_');

else {

while (!\_operators\_stack.empty() && (\_operators\_stack.top() == '\_' || operators\_priority(entered\_string[i]) >= operators\_priority(\_operators\_stack.top()))) {

expression.push\_back(\_operators\_stack.top());

expression.push\_back(' ');

\_operators\_stack.pop();

}

\_operators\_stack.push(entered\_string[i]);

}

}

while (!\_operators\_stack.empty()) {

expression.push\_back(\_operators\_stack.top());

expression.push\_back(' ');

\_operators\_stack.pop();

}

return expression;

}

double arithmetic::compute(const std::string& postfix\_expression) {

stack<double> \_operands\_stack;

for (size\_t i = 0; i < postfix\_expression.length(); ++i) {

if (is\_digit(postfix\_expression[i]) || is\_dot(postfix\_expression[i])) {

double temp = 0;

for (; i < postfix\_expression.length() && is\_digit(postfix\_expression[i]); ++i)

temp = temp \* 10 + (postfix\_expression[i] - '0');

if (postfix\_expression[i] == '.') {

double divide = 1;

for (i++; i < postfix\_expression.length() && is\_digit(postfix\_expression[i]); ++i) {

temp = temp \* 10 + (postfix\_expression[i] - '0');

divide \*= 10;

}

temp /= divide;

}

\_operands\_stack.push(temp);

--i;

}

else if (is\_operator(postfix\_expression[i])) {

double operand\_1 = \_operands\_stack.pop();

double operand\_2 = \_operands\_stack.pop();

if (postfix\_expression[i] == '+')

\_operands\_stack.push(operand\_1 + operand\_2);

if (postfix\_expression[i] == '-')

\_operands\_stack.push(operand\_2 - operand\_1);

if (postfix\_expression[i] == '\*')

\_operands\_stack.push(operand\_1 \* operand\_2);

if (postfix\_expression[i] == '/')

\_operands\_stack.push(operand\_2 / operand\_1);

}

else if (postfix\_expression[i] == '\_')

\_operands\_stack.push(\_operands\_stack.pop() \* (-1));

}

return \_operands\_stack.pop();

}