1. Министерство высшего образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 2**

«Калькулятор»

1. по дисциплине «Структуры данных»
2. Выполнил
3. студент гр. Белоконь Д. А., Солуянов А. Д.
4. <*подпись*>

Проверил Вагисаров В. Б.

1. <*подпись*>

1. Санкт-Петербург
2. 2023
3. **Цель работы**

Необходимо написать программу, реализующую калькулятор.

Формулировка задания

Нужно реализовать калькулятор, поддерживающий арифметические операции и функции сложения, вычитания, умножения, деления, возведения в степень, факториала, синуса, косинуса и корня. Также должны поддерживаться переменные. Программа должна быть построена на основе алгоритма подсчёта значения в постфиксной нотации и алгоритма сортировочной станции с использованием структур данных стек и очередь.

Результаты

Сначала реализуем структуру данных стека и его функций. Вводим два поля: void \*data и struct Stack \*nextElement, а также пишем функции createStack, pop, top, push, removeElement.

Затем пишем недостающую к math.h функцию факториала.

После идёт сам main, где парсим аргументы (если есть) или считываем строку. Затем парсим строку, удаляя пробелы и заменяя тригонометрические функции на их односимвольные обозначения. Ну а затем идёт по строке, распределяя числа, операторы и переменные по обычной схеме.

Вывод

Это мой пятый польский калькулятор на четвёртом языке программирования. Я сделал уже все выводы, которые мог.

Приложение

main.c:

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include "stack.h"

#include "math.h"

char newLevel;

int isoperator(char c) {

return c == '+' || c == '-' || c == '\*' || c == '/' || c == '^' || c == '(' || c == ')' ||

c == '!' || c == '$' || c == '#' || c == '&' || c == '%';

}

int funccmp(char a, char b) {

char prioritisation[256];

prioritisation['+'] = 1;

prioritisation['-'] = 1;

prioritisation['\*'] = 2;

prioritisation['/'] = 2;

prioritisation['!'] = 3;

prioritisation['^'] = 3;

prioritisation['%'] = 3;

prioritisation['$'] = 3;

prioritisation['#'] = 3;

prioritisation['&'] = 3;

return prioritisation[(int)a] - prioritisation[(int)b];

}

void handleOperator(char c, struct Stack \*operators, struct Stack \*numbers) {

char \*newOperator = malloc(sizeof(char) \* 1);

newOperator[0] = c;

if (c == '(') {

push(operators, newOperator);

return;

}

struct Stack \*previousElement = top(operators);

if (previousElement->data == NULL) {

push(operators, newOperator);

return;

}

char previousOperator = \*((char \*)(previousElement->data));

if (previousOperator == '(') {

if (c == ')') {

previousElement = pop(operators);

removeElement(previousElement);

return;

}

push(operators, newOperator);

return;

}

if (c != ')' && c != '\0' && funccmp(c, previousOperator) > 0) {

push(operators, newOperator);

return;

}

previousElement = pop(operators);

struct Stack \*firstElement;

struct Stack \*secondElement;

double firstNumber;

double secondNumber;

double \*result = (double \*)malloc(sizeof(double));

switch (previousOperator) {

case '+':

firstElement = pop(numbers);

secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

\*result = secondNumber + firstNumber;

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

case '-':

firstElement = pop(numbers);

secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

\*result = secondNumber - firstNumber;

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

case '\*':

firstElement = pop(numbers);

secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

\*result = secondNumber \* firstNumber;

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

case '/':

firstElement = pop(numbers);

secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

\*result = (double)(secondNumber / firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

case '^':

firstElement = pop(numbers);

secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

\*result = pow(secondNumber, firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

case '!':

firstElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

\*result = factorial(firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(previousElement);

break;

case '$':

firstElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

\*result = sqrt(firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(previousElement);

break;

case '#':

firstElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

\*result = sin(firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(previousElement);

break;

case '&':

firstElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

\*result = cos(firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(previousElement);

break;

case '%':

firstElement = pop(numbers);

struct Stack \*secondElement = pop(numbers);

firstNumber = \*((double \*)(firstElement->data));

secondNumber = \*((double \*)(secondElement->data));

if (firstNumber / 1 != firstNumber || secondNumber / 1 != secondNumber) {

fprintf(stderr, "Float number is given to modulus operator.\n");

exit(1);

}

\*result = (double)((int)secondNumber % (int)firstNumber);

push(numbers, result);

removeElement(firstElement);

removeElement(secondElement);

removeElement(previousElement);

break;

default:

fprintf(stderr, "Unknown operator: %c\n", c);

exit(1);

}

if (c != '\0')

handleOperator(c, operators, numbers);

}

void parseInput(char \*input) {

char newInput[strlen(input) \* 2 + 1];

char \*word;

while ((word = strstr(input, "sin")) != NULL) {

word[0] = '#';

memcpy(newInput, input, (word - input) + 1);

memcpy(newInput + (word - input) + 1, word + 3, strlen(word + 3) + 1);

memcpy(input, newInput, strlen(newInput) + 1);

}

while ((word = strstr(input, "cos")) != NULL) {

word[0] = '&';

memcpy(newInput, input, (word - input) + 1);

memcpy(newInput + (word - input) + 1, word + 3, strlen(word + 3) + 1);

memcpy(input, newInput, strlen(newInput) + 1);

}

while ((word = strstr(input, "sqrt")) != NULL) {

word[0] = '$';

memcpy(newInput, input, (word - input) + 1);

memcpy(newInput + (word - input) + 1, word + 4, strlen(word + 4) + 1);

memcpy(input, newInput, strlen(newInput) + 1);

}

while ((word = strstr(input, " ")) != NULL ||

(word = strstr(input, "\n"))!= NULL) {

memcpy(newInput, input, (word - input));

memcpy(newInput + (word - input), word + 1, strlen(word + 1) + 1);

memcpy(input, newInput, strlen(newInput) + 1);

}

while ((word = strstr(input, "-(")) != NULL) {

memcpy(newInput, input, (word - input) + 1);

memcpy(newInput + (word - input) + 3, word + 1, strlen(word + 1) + 1);

newInput[word - input + 1] = '1';

newInput[word - input + 2] = '\*';

memcpy(input, newInput, strlen(newInput) + 1);

}

}

int main(int argc, char \*argv[]) {

newLevel = 1;

char \*input;

if (argc < 2) {

printf("Not enough arguments, please, enter a line:\n");

input = (char \*)malloc(sizeof(char) \* 4076);

fgets(input, 4076, stdin);

} else {

int inputSize = 0;

for (int i = 1; i < argc; i++)

inputSize += strlen(argv[i]);

input = (char \*)malloc(sizeof(char) \* inputSize \* 2);

for (int i = 1; i < argc; i++)

strcpy(input, argv[i]);

input[inputSize - 1] = '\0';

}

parseInput(input);

struct Stack \*numbers = createStack();

struct Stack \*operators = createStack();

for (int i = 0; i < strlen(input) + 1; i++) {

printf("%c = %d\n", input[i], input[i]);

}

for (int i = 0; i < strlen(input) + 1; i++) {

if (isdigit(input[i]) || (input[i] == '-' && newLevel)) {

double \*newNumber = (double \*)malloc(sizeof(double));

char \*endPointer;

\*newNumber = strtod(input + i, &endPointer);

i += endPointer - (input + i) - 1;

push(numbers, newNumber);

newLevel = 0;

} else if (input[i] == '\0' || isoperator(input[i])) {

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

newLevel = 1;

}

if (input[i] == '\0') {

while (operators->data != NULL)

handleOperator(input[i], operators, numbers);

} else {

handleOperator(input[i], operators, numbers);

}

} else {

printf("Enter a number for variable %c: ", input[i]);

fflush(stdout);

fflush(stdin);

double \*newNumber = (double \*)malloc(sizeof(double));

scanf("%lf", newNumber);

push(numbers, newNumber);

newLevel = 0;

}

}

printf("%f\n", \*((double \*)(pop(numbers)->data)));

return 0;

}

Factorial.c:

#include "math.h"

double factorial(int a) {

double result = a;

while (--a)

result \*= a;

return result;

}

Stack.c:

#include <stdlib.h>

#include <string.h>

#include "stack.h"

struct Stack \*createStack(void) {

struct Stack \*newStack = (struct Stack \*)malloc(sizeof(struct Stack) \* 1);

newStack->data = NULL;

newStack->nextElement = NULL;

return newStack;

}

struct Stack \*top(struct Stack \*stack) {

while (stack->nextElement != NULL)

stack = stack->nextElement;

return stack;

}

struct Stack \*pop(struct Stack \*stack) {

struct Stack \*nextElement = stack->nextElement;

if (nextElement == NULL) {

struct Stack \*copyStack = (struct Stack \*)malloc(sizeof(struct Stack));

memcpy(copyStack, stack, sizeof(struct Stack));

stack->data = NULL;

stack->nextElement = NULL;

return copyStack;

}

while (nextElement->nextElement != NULL) {

stack = nextElement;

nextElement = stack->nextElement;

}

stack->nextElement = NULL;

return nextElement;

}

void push(struct Stack \*stack, void \*data) {

if (stack->data == NULL) {

stack->data = data;

return;

}

if (stack->nextElement != NULL)

stack = top(stack);

struct Stack \*newElement = createStack();

newElement->data = data;

stack->nextElement = newElement;

return;

}

void removeElement(struct Stack \*stack) {

if (stack == NULL)

return;

if (stack->data != NULL) {

free(stack->data);

return;

}

free(stack->data);

return;

}