Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет
телекоммуникаций и информатики»

Факультет ИВТ

Кафедра вычислительных систем

Курсовая работа

на тему «Разработка инструментов командной строки ОС GNU/LINUX»
Вариант 2.2 «Калькулятор командной строки»

Выполнил:
студент гр. ИВ-222
Николаенков М.Д.

Проверил:
Ст. Преподаватель Кафедры ВС
Фульман В.О.

Новосибирск, 2023

ОГЛАВЛЕНИЕ

ЗАДАНИЕ

Тема курсовой работы
«Разработка инструментов командной строки ОС GNU/LINUX»
Вариант 2.2 «Калькулятор командной строки»

Задание на курсовую работу
Разработать программу-калькулятор **cmdcalc**, предназначенную для вычисления простейших арифметических выражений с учетом приоритета операций и расстановки скобок.
На вход команды **cmdcalc** через аргументы командной строки поступает символьная строка, содержащая арифметическое выражение.
Требуется проверить корректность входного выражения (правильность расстановки операндов, операций и скобок) и, если выражение корректно, вычислить его значение.

Арифметическое выражение записывается следующим образом: A p B или (A p B), где A – левый операнд, B - правый операнд, p – арифметическая операция.
A и B – представляют собой арифметические выражения или вещественные числа, p = + | - | * | /. Например: ((((1.1-2) +3) * (2.5*2)) + 10), (1.1 – 2) +3 * (2.5 * 2) + 10.
Пример вызова команды **cmdcalc** (обратите внимание, что входное выражение необходимо взять в кавычки!): $ **cmdcalc** "3 * 2 – 1 + 3" Полученный ответ может отображаться на экране, а также сохраняться в файле.

Критерии оценки
Оценка «удовлетворительно»: не предусмотрены скобки и приоритеты операций.
Оценка «хорошо»: учитываются приоритеты операций.
Оценка «отлично»: учитываются приоритеты операций, допускаются скобки.

ВЫПОЛНЕНИЕ РАБОТЫ

Описывается ход работы над заданием с приложением снимков экрана;

Анализ задачи

1. В данной работе требуется разработать калькулятор командной строки. Все входные данные передаются через аргументы командной строки (с использованием аргументов функции main.
   Для создания такого алгоритма, необходимо реализовать стек для хранения чисел и знаков, также нужно учитывать приоритеты знаков для того, чтобы калькулятор считал правильно, (для + и - => это 1, а для * и / => это 2) дабы вначале происходило деление и умножение, а потом уже сложение и вычитание.
   Смысл алгоритма довольно прост - мы достаем из стека 2 числа и символ, и выполняем операцию между ними, после которой возвращаем результат в стек с числами. (если строка закончилась то до повторяем это пока не закончились символы или пока приоритет символа меньше или равен текущему приоритету символа в стеке)
   Входная строка проверяется в функции check, которая возвращает 0 или 1 в зависимости от корректности входной строки, после чего, в зависимости от результата будет выполняться сам алгоритм - main_program, который будет возвращать число, которое получилось в результате выполнения операций.

2. В данной работе были реализованы следующие функции:

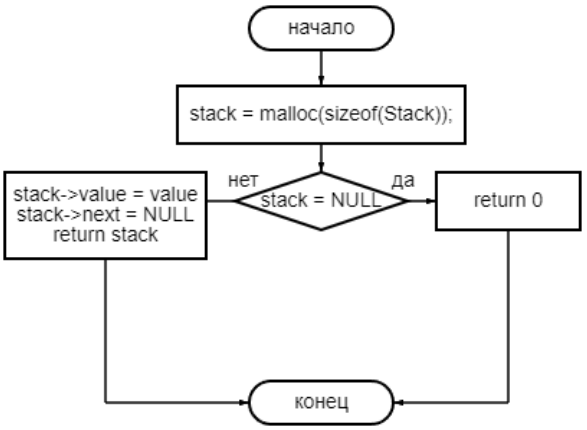
Header.h

```
1   int check(char* str)
2   int priority(char symbol)
3   double match(double a, double b, char s)
4   void process(StackDouble** stack_d, StackChar** stack_c)
5   double main_program(char* str)
```

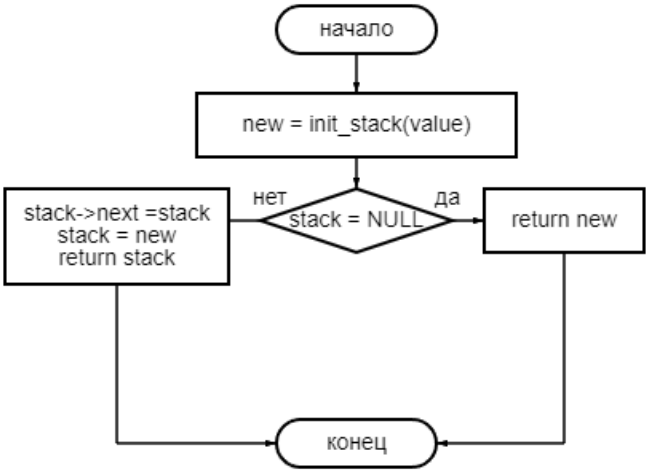
Stacks.h

```
1   StackDouble* init_stack_double(double value);
2   StackDouble* push_double(StackDouble* stack, double value);
3   double pop_double(StackDouble** stack);
4   int is_empty_double(StackDouble* st);
5   double get_double(StackDouble* st);
6   StackChar* init_stack_char(char value);
7   StackChar* push_char(StackChar* stack, char value);
8   char pop_char(StackChar** stack);
9   int is_empty_char(StackChar* st);
10  char get_char(StackChar* st);
```

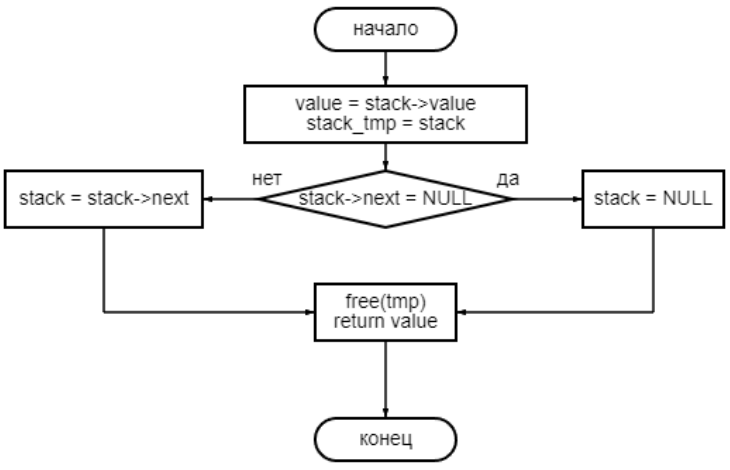
StackDouble* init_stack_double(double value)
StackChar* init_stack_char(char value)



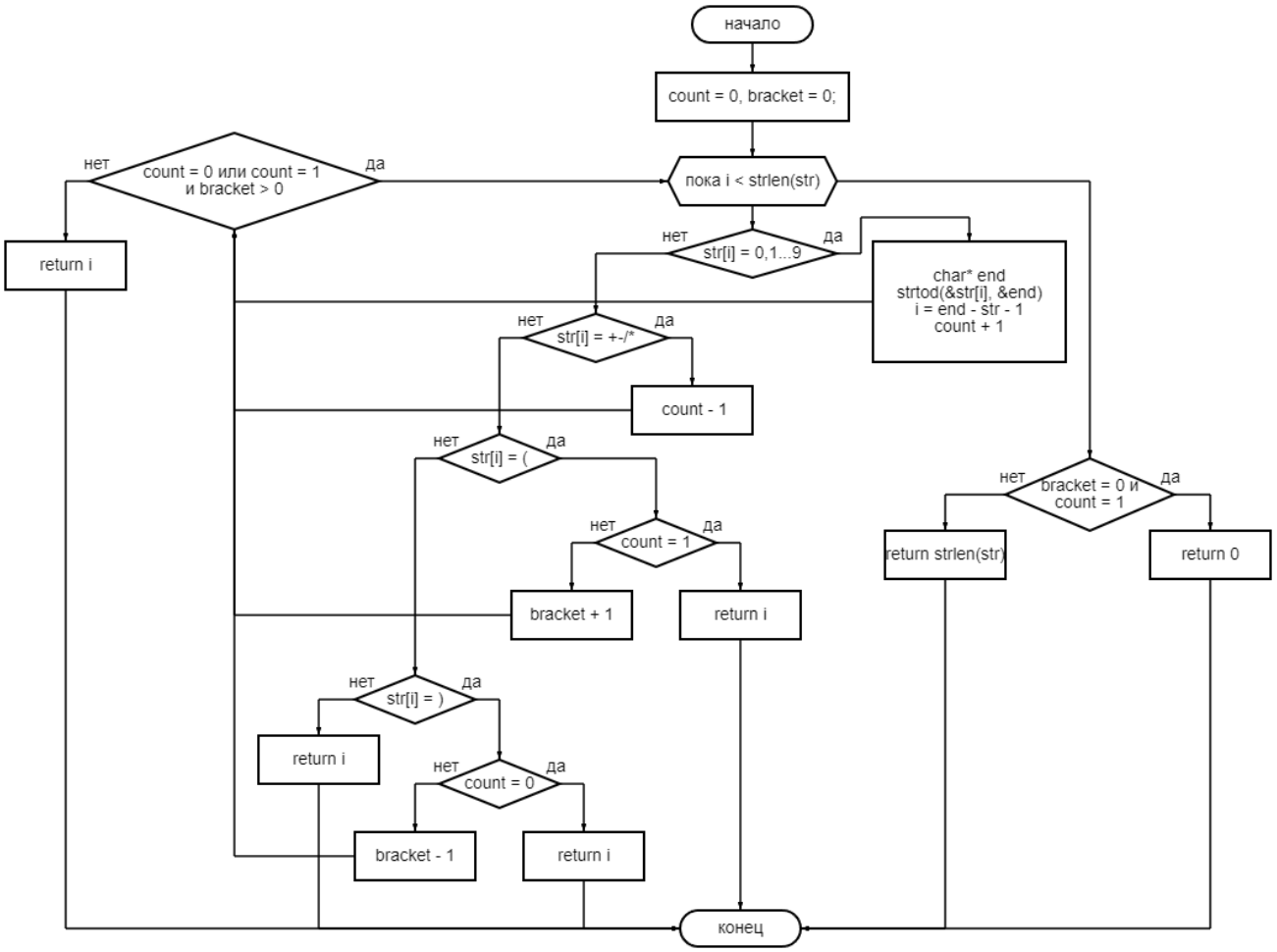
StackDouble* push_double(StackDouble* stack, double value)
StackChar* push_char(StackChar* stack, char value)



double pop_double(StackDouble** stack)
char pop_char(StackChar** stack)

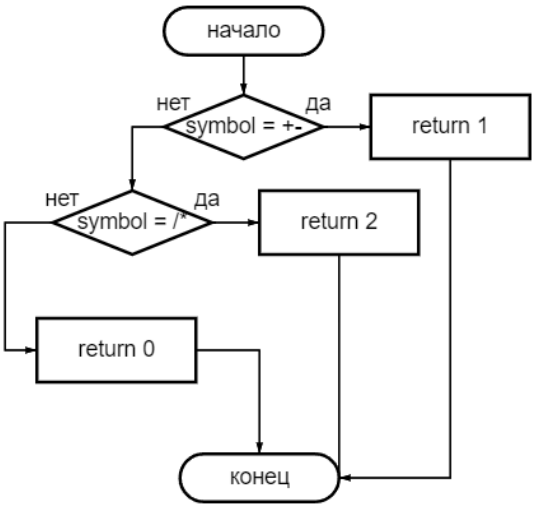


int check(char* str)



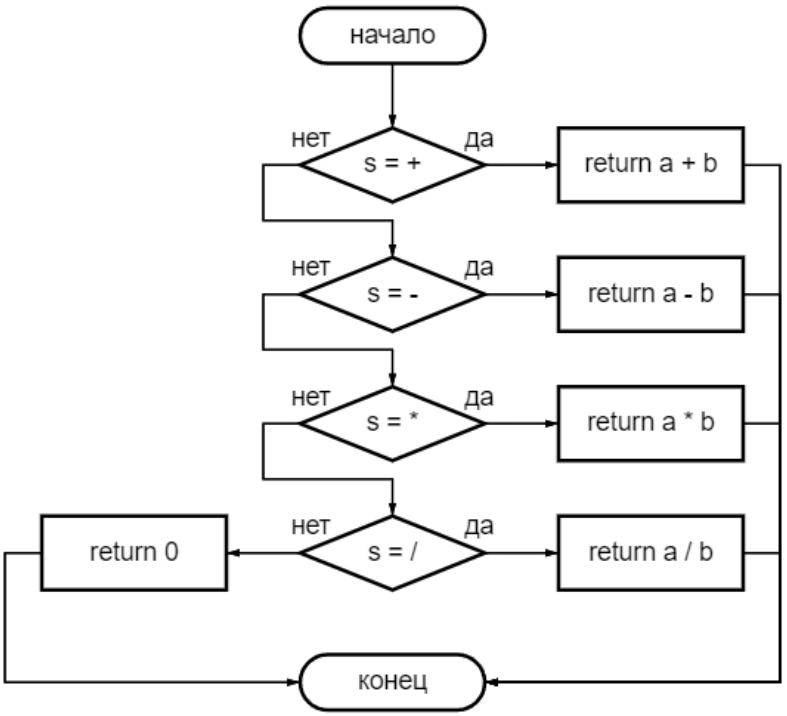
Эта функция принимает на вход указатель на начало строки, использует 2 переменные для проверки корректности последовательности вида (A p B) и правильности расставления скобок: count и bracket и возвращает значение 0 в случае успеха, в случае ошибки, возвращает индекс элемента, на котором программа словила ошибку.
Каждый раз, когда в строке встречается число, переменная count увеличивается на 1, а когда встречает символ (+-/*), то уменьшается на 1, в результате мы получим переменную, которая меняется в диапазоне от 0 до 1, если этот диапазон будет отличаться, то это будет означать, что последовательность имеет неправильный вид. В конце эта переменная должна быть равна 1 что означает, что в конце - число.
Со скобками схожая ситуация, переменная bracket увеличивается на 1 если видит открытую скобку или уменьшается на 1, если видит закрытую. В конце переменная должна быть равна 0, что означает, что скобки все закрылись.
Также, если функция видит незнакомый символ, она возвращает индекс этого символа.

int priority(char symbol)



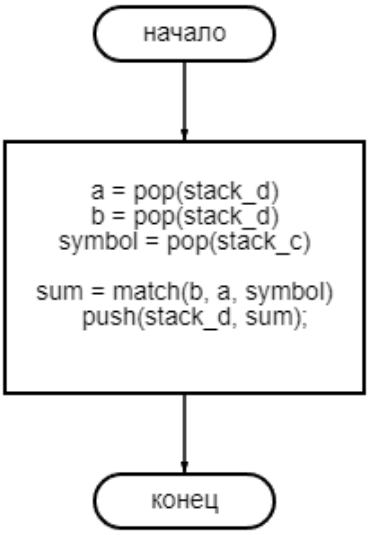
Данная функция на вход принимает символ арифметической операции и возвращает ее приоритет.
Функция проверяет переданный символ операции. Если символ равен '+' или '-', функция возвращает 1, если символ равен '*' или '/', функция возвращает 2. В остальных случаях функция возвращает 0.
Таким образом, функция используется для определения приоритета операции.

double match(double a, double b, char s)



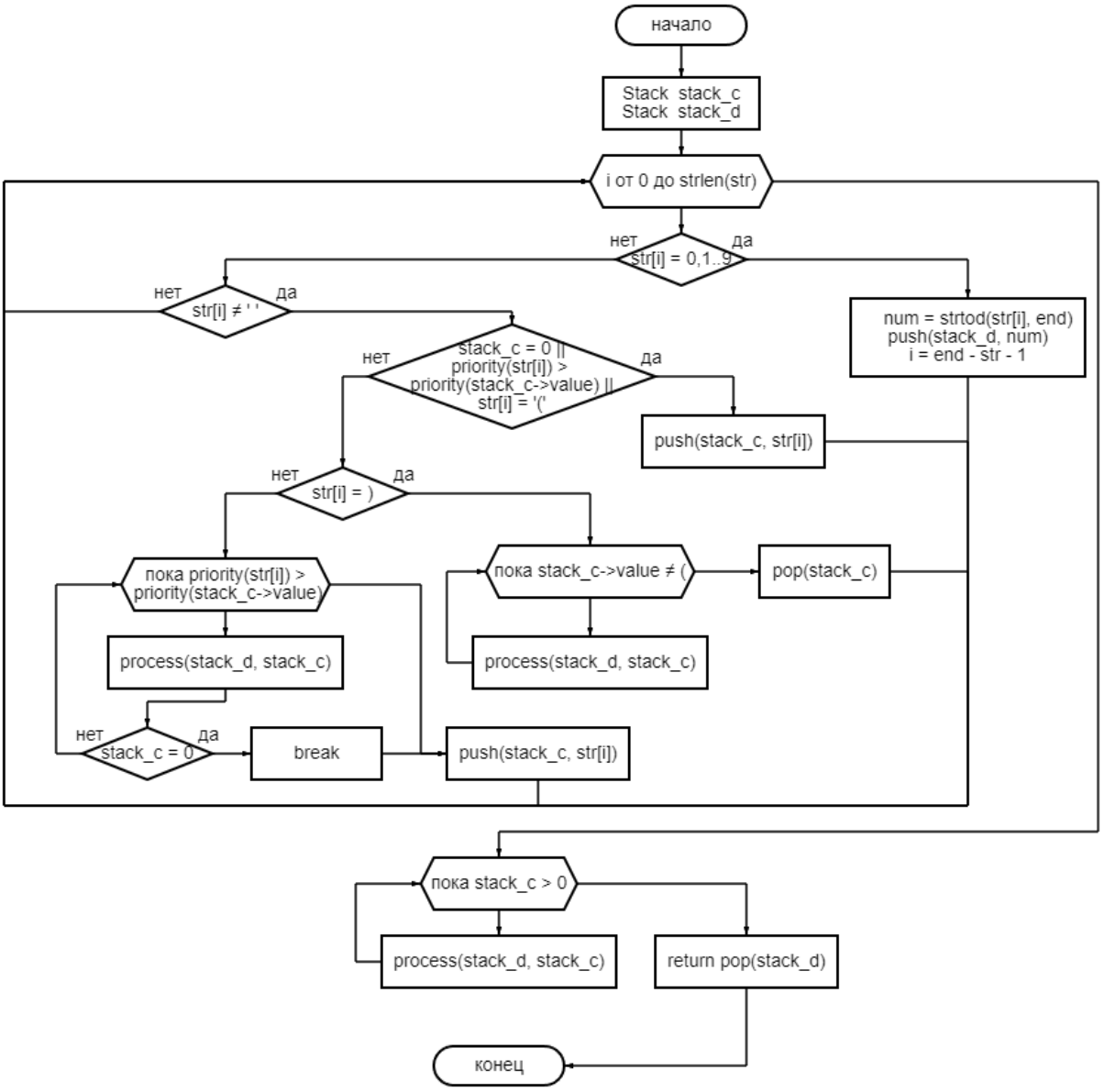
Эта функция принимает на вход три аргумента: два числа типа double (a и b) и символ s. Функция возвращает результат операции, определенной символом s, между числами a и b.
Если символ s равен '+', функция возвращает сумму a + b. Если символ s равен '-', функция возвращает разность a - b. Если символ s равен '*', функция возвращает произведение a * b. Если символ s равен '/', функция возвращает частное a / b. Если символ s не соответствует ни одному из этих символов ('+', '-', '*' или '/'), функция возвращает 0.

```
void process(StackDouble** stack_d, StackChar** stack_c)
```



Функция process() извлекает из стека - два значения типа double и одно значение типа char, и сохраняет их в переменные a, b и symbol соответственно. Затем она вызывает функцию match(), передавая ей значения a, b и symbol, чтобы выполнить операцию, определенную символом. Результат операции помещается обратно в стек со значениями типа double.

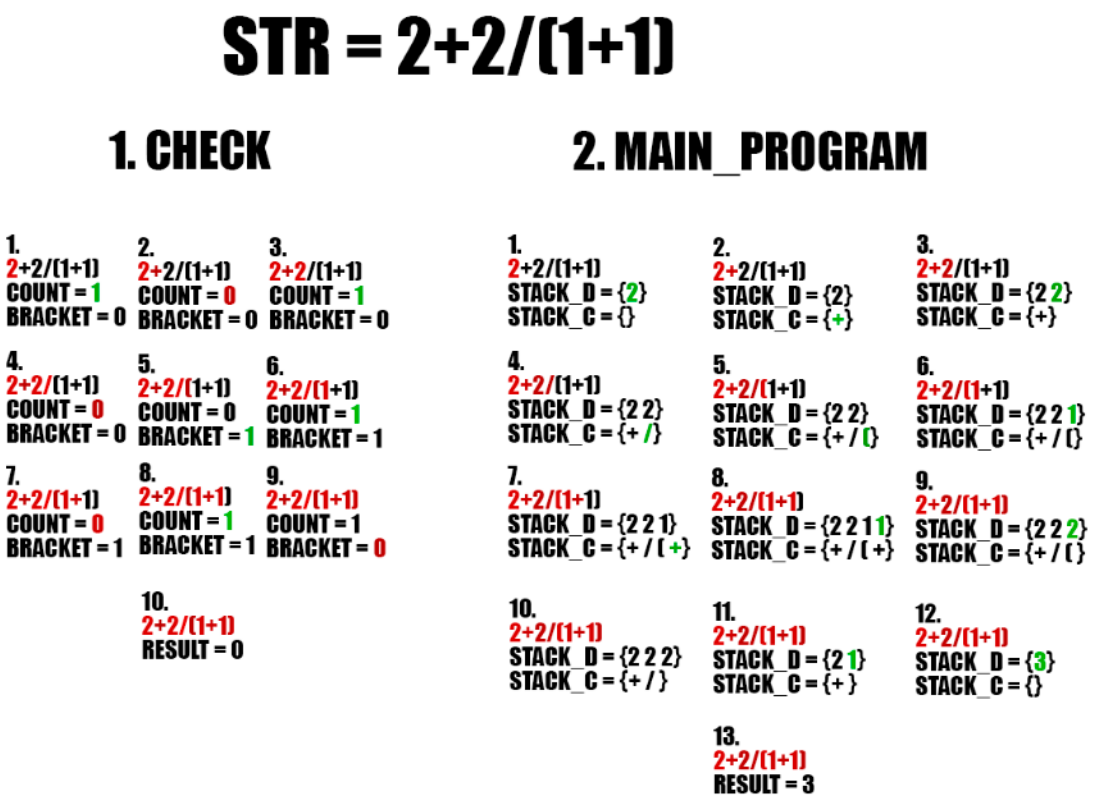
double main_program(char* str)



Данная функция принимает на вход строку str, содержащую математическое выражение и возвращает результат его вычисления в виде числа типа double. Алгоритм использует два стека: stack_d, для хранения чисел, и stack_c, для хранения операторов. Он проходит по символам входной строки, и в зависимости от типа символа, выполняет определенные действия:

- Если символ является цифрой, то происходит извлечение числа из строки и добавление его в стек stack_d.
- Если символ является оператором или скобкой, то он добавляется в стек stack_c. При этом, если текущий оператор имеет больший приоритет, чем оператор на вершине стека stack_c, то он также добавляется в стек. Если текущий символ является закрывающей скобкой, то происходит выполнение операций из стека stack_c до тех пор, пока не будет достигнута соответствующая открывающая скобка.
- После обработки всех символов входной строки, происходит выполнение всех оставшихся операций из стека stack_c до его полной очистки.

Пошаговая работа алгоритма:



Тестовые данные
“5 * (5 - 3) ” = 10
“25 + 5 * 10 ” = 75

“25 + 10 + ” = Error
“25 +) 10 ” = Error

Результаты тестирования разработанной программы:

```
[vladimir@fedora bin]$ valgrind ./cmdcalc "5 * (5 - 3)" "25 + 5 * 10" "25 + 10 +" "25 +  ) 10"
==23547== Memcheck, a memory error detector
==23547== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==23547== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==23547== Command: ./cmdcalc 5\ *\ (5\ -\ 3) 25\ +\ 5\ *\ 10 25\ +\ 10\ + 25\ +\ \ )\ 10
==23547==
5 * (5 - 3) = 10.00
25 + 5 * 10 = 75.00
25 + 10 +
         ^
ERROR
25 +  ) 10
      ^
ERROR
==23547==
==23547== HEAP SUMMARY:
==23547==     in use at exit: 0 bytes in 0 blocks
==23547==   total heap usage: 16 allocs, 16 frees, 1,264 bytes allocated
==23547==
==23547== All heap blocks were freed -- no leaks are possible
==23547==
==23547== For lists of detected and suppressed errors, rerun with: -s
==23547== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

ПРИЛОЖЕНИЕ

cmdcalc.c

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <libcmdcalc/header.h>

int main(int argc, char** argv)
{
    for (int arg = 1; arg < argc; arg++) {
        char* str = argv[arg];
        int status = check(str);
        if (status) {
            print_error(str, status);
            continue;
        }
        else{
            double result = main_program(str);
            printf("%s = %.2f\n", str, result);
        }
    }
    return 0;
}
```

header.h

```c
#pragma once

#include <libcmdcalc/stacks.h>

int check(char* str);
int priority(char symbol);
double match(double a, double b, char s);
void process(StackDouble** st_double, StackChar** st_char);
double main_program(char* str);
void print_error(char* str, int id);
```

stacks.h

```c
#pragma once

typedef struct StackDouble {
    double value;
    struct StackDouble* next;
} StackDouble;

typedef struct StackChar {
    char value;
    struct StackChar* next;
} StackChar;

StackDouble* init_stack_double(double value);
StackDouble* push_double(StackDouble* stack, double value);
double pop_double(StackDouble** stack);
int is_empty_double(StackDouble* st);
```

```
17      double get_double(StackDouble* st);
18
19      StackChar* init_stack_char(char value);
20      StackChar* push_char(StackChar* stack, char value);
21      char pop_char(StackChar** stack);
22      int is_empty_char(StackChar* st);
23      char get_char(StackChar* st);
```

header.c

```
1       #include <stdint.h>
2       #include <stdio.h>
3       #include <stdlib.h>
4       #include <string.h>
5
6       #include <libcmdcalc/header.h>
7       #include <libcmdcalc/stacks.h>
8
9       int check(char* str)
10      {
11          int count = 0, bracket = 0;
12          for (size_t i = 0; i < strlen(str); i++) {
13              if (str[i] == ' ')
14                  continue;
15
16              if ((str[i] >= '0' && str[i] <= '9')) {
17                  char* end;
18                  strtod(&str[i], &end);
19                  i = end - str - 1;
20                  count += 1;
21              } else if (
22                      str[i] == '/' || str[i] == '*' || str[i] == '+'
23                      || str[i] == '-') {
24                  count -= 1;
25              } else if (str[i] == '(') {
26                  if (count == 1)
27                      return (int)i;
28                  bracket += 1;
29              } else if (str[i] == ')') {
30                  if (count == 0)
31                      return (int)i;
32                  bracket -= 1;
33              } else
34                  return (int)i;
35
36              if (count < 0 || count > 1 || bracket < 0) {
37                  return i;
38              }
39          }
40          if (bracket == 0 && count == 1)
41              return 0;
42          else
43              return (int)strlen(str);
44      }
45
46      int priority(char symbol)
47      {
48          switch (symbol) {
49          case '+':
50          case '-':
51              return 1;
52          case '*':
53          case '/':
54              return 2;
```

```
55
56      default:
57          return 0;
58      }
59  }
60
61  double match(double a, double b, char s)
62  {
63      switch (s) {
64      case '+':
65          return a + b;
66      case '-':
67          return a - b;
68      case '*':
69          return a * b;
70      case '/':
71          return a / b;
72      default:
73          return 0;
74      }
75  }
76
77  void process(StackDouble** st_double, StackChar** st_char)
78  {
79      double a = pop_double(st_double);
80      double b = pop_double(st_double);
81      char symbol = pop_char(st_char);
82      *st_double = push_double(*st_double, match(b, a, symbol));
83  }
84
85  double main_program(char* str)
86  {
87      StackDouble* st_double = NULL;
88      StackChar* st_char = NULL;
89      for (size_t i = 0; i < strlen(str); i++) {
90          if (str[i] >= '0' && str[i] <= '9') {
91              char* end;
92              double num = strtod(&str[i], &end);
93              i = end - str - 1;
94              st_double = push_double(st_double, num);
95
96          } else if (str[i] != ' ') {
97              if (st_char == NULL || priority(str[i]) > priority(st_char->value)
98                  || str[i] == '(') {
99                  st_char = push_char(st_char, str[i]);
100             } else if (str[i] == ')') {
101                 while (get_char(st_char) != '(') {
102                     process(&st_double, &st_char);
103                 }
104                 pop_char(&st_char);
105             } else {
106                 while (priority(str[i]) <= priority(get_char(st_char))) {
107                     process(&st_double, &st_char);
108                     if (is_empty_char(st_char))
109                         break;
110                 }
111                 st_char = push_char(st_char, str[i]);
112             }
113         }
114     }
115     while (!is_empty_char(st_char)) {
116         process(&st_double, &st_char);
117     }
118
119     return pop_double(&st_double);
```

```
120    }
121
122    void print_error(char* str, int id)
123    {
124        printf("%s\n", str);
125        for (int i = 0; i < id; i++) {
126            printf(" ");
127        }
128        printf("^\nERROR\n");
129    }
```

stacks.c

```
 1    #include <stdint.h>
 2    #include <stdio.h>
 3    #include <stdlib.h>
 4
 5    #include <libcmdcalc/stacks.h>
 6
 7    StackDouble* init_stack_double(double value)
 8    {
 9        StackDouble* stack = (StackDouble*)malloc(sizeof(StackDouble));
10        if (!stack)
11            return NULL;
12        stack->value = value;
13        stack->next = NULL;
14        return stack;
15    }
16
17    StackDouble* push_double(StackDouble* stack, double value)
18    {
19        StackDouble* new = init_stack_double(value);
20        if (!new)
21            return NULL;
22        if (!stack) {
23            stack = new;
24            return stack;
25        }
26        new->next = stack;
27        stack = new;
28        return stack;
29    }
30
31    double pop_double(StackDouble** stack)
32    {
33        if (!(*stack))
34            return -1;
35
36        double value = (*stack)->value;
37        StackDouble* tmp = *stack;
38
39        if (!(*stack)->next) {
40            *stack = NULL;
41        } else {
42            *stack = (*stack)->next;
43        }
44
45        free(tmp);
46        return value;
47    }
48
49    int is_empty_double(StackDouble* st){
```

```
50      if(!st)
51          return 1;
52      return 0;
53  }
54
55  double get_double(StackDouble* st){
56      return st->value;
57  }
58  StackChar* init_stack_char(char value)
59  {
60      StackChar* stack = (StackChar*)malloc(sizeof(StackChar));
61      if (!stack)
62          return NULL;
63      stack->value = value;
64      stack->next = NULL;
65      return stack;
66  }
67
68  StackChar* push_char(StackChar* stack, char value)
69  {
70      StackChar* new = init_stack_char(value);
71      if (!new)
72          return NULL;
73      if (!stack) {
74          stack = new;
75          return stack;
76      }
77      new->next = stack;
78      stack = new;
79      return stack;
80  }
81
82  char pop_char(StackChar** stack)
83  {
84      if (!(*stack))
85          return '\0';
86
87      char value = (*stack)->value;
88      StackChar* tmp = *stack;
89
90      if (!(*stack)->next) {
91          *stack = NULL;
92      } else {
93          *stack = (*stack)->next;
94      }
95
96      free(tmp);
97      return value;
98  }
99
100 int is_empty_char(StackChar* st){
101     if(!st)
102         return 1;
103     return 0;
104 }
105
106 char get_char(StackChar* st){
107     return st->value;
108 }
```