**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

Курсовая РАБОТА

**по дисциплине «Программирование»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1302 |  | Новиков Г.В. |
| Преподаватель |  | Калмычков В.А |

Санкт-Петербург

2022

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент: Новиков Г.В. | | |
| Группа 1302 | | |
| Исходные данные: Текст, представляющий собой последовательность строк, длина каждой из которых не превышает некоторого задаваемого при выполнении значения, вводится с клавиатуры или из файла.  Необходимо: преобразовать текст в соответствии с последовательностью команд редактирования, которые должны позволять вставлять, удалять и заменять заданные слова и подстроки | | |
| Содержание пояснительной записки: цель курсовой работы, исходная постановка, способы реализации ввода-вывода, написанные функции, разбиение программы на файлы, алгоритмы и структуры данных, текст программы, тестирование программы, вывод. | | |
| Дата выдачи задания: | | |
| Дата сдачи реферата: | | |
| Дата защиты реферата: | | |
| Студент |  | Новиков Г.В. |
| Преподаватель |  | Калмычков В.А |

**Аннотация**

В работе представлена программа, решающая поставленную задачу по редактированию заданного текста. Входные данные представляют собой либо файл, либо введенный с клавиатуры текст. По команде пользователя осуществляется одна из операций: удаление, вставка или замена набора символов, определяемых понятиями контекста и номера текущей строки, а также вывод отредактированного текста на экран, переход между строками и запись результата в указываемый файл.

**Summary**

The paper presents a program that solves the task of editing a given text. The input data is either a file or text entered from the keyboard. At the user's command, one of the operations is performed: deletion, insertion or replacement of a set of characters defined by the concepts of the context and current line number, as well as displaying the edited text on the screen, switching between lines and writing the result to the specified file.

Оглавление

[Цель курсовой работы: 5](#_Toc105509868)

[Исходная постановка: 5](#_Toc105509869)

[Способы реализации ввода-вывода 5](#_Toc105509870)

[Написанные функции: 5](#_Toc105509871)

[Разбиение программы на файлы 8](#_Toc105509872)

[Алгоритмы и структуры данных 9](#_Toc105509873)

[Текст программы 9](#_Toc105509874)

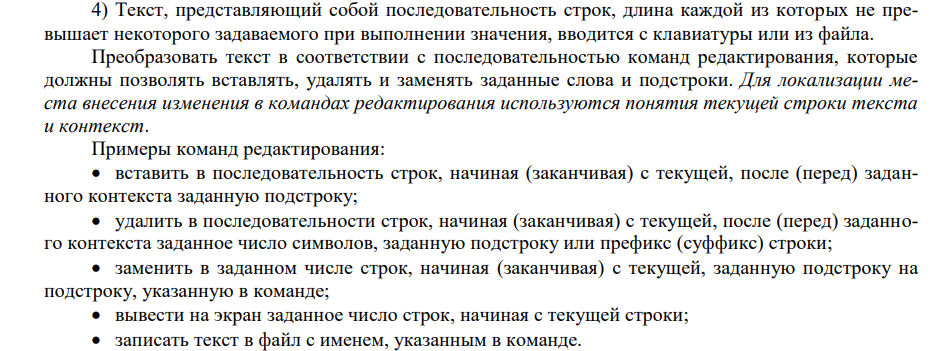
[Тестирование программы 16](#_Toc105509875)

[Вывод 18](#_Toc105509876)

# **Цель курсовой работы:**

Продемонстрировать умение хранить, обрабатывать текстовыми данными, представленные в виде связных списков

# Исходная постановка:



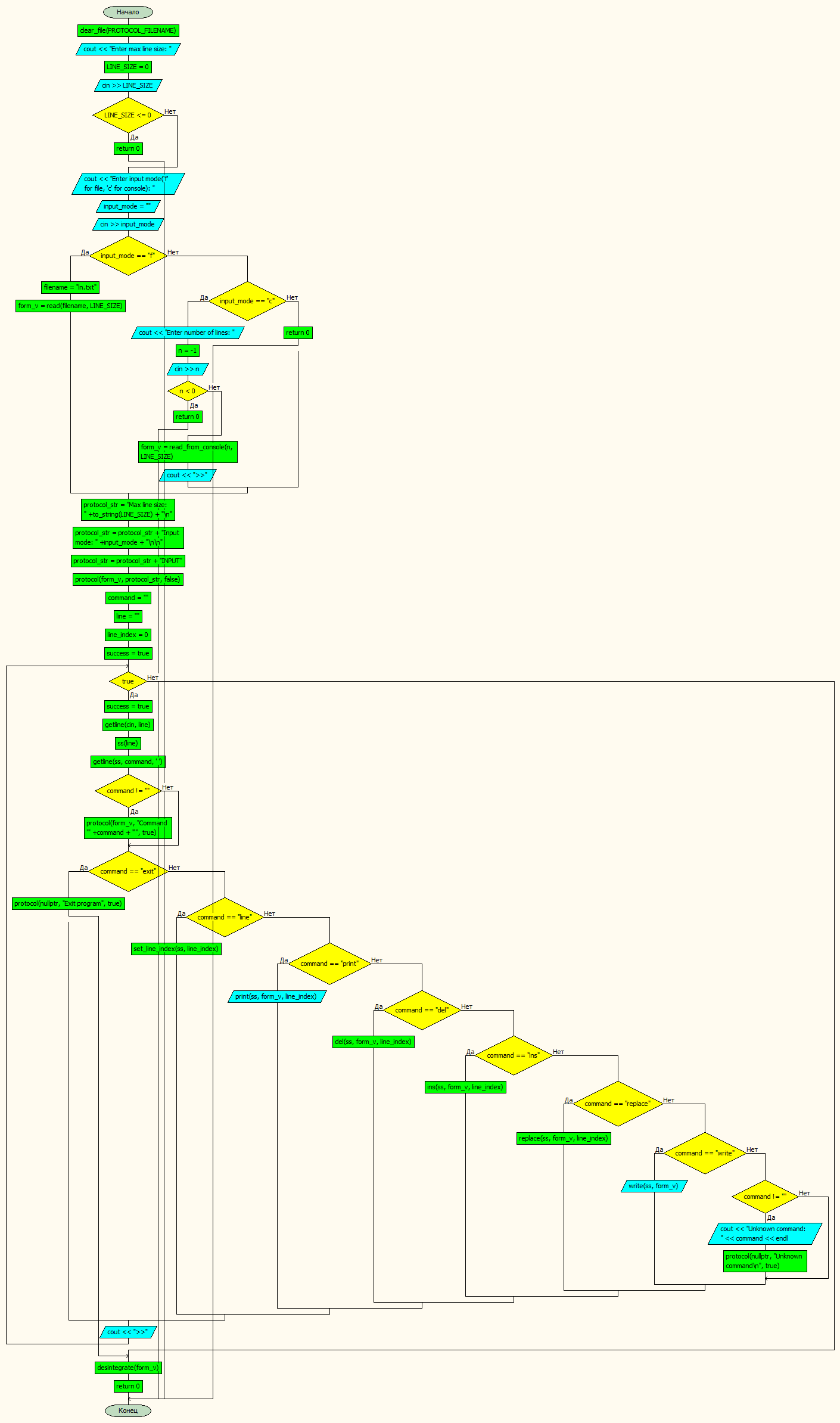
Способы реализации ввода-вывода:

|  |  |  |  |
| --- | --- | --- | --- |
|  | iostream | fstream | string |
| Ввод | cin | ios::in | getline |
| Вывод | cout | ios::out |  |

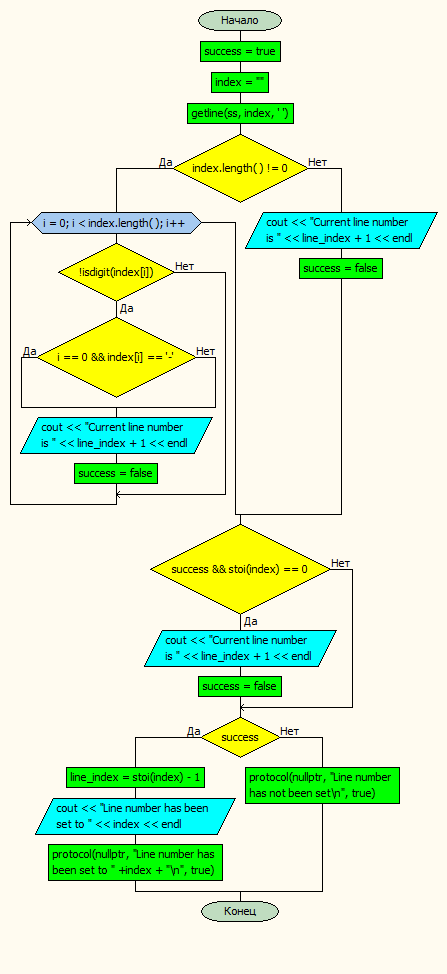
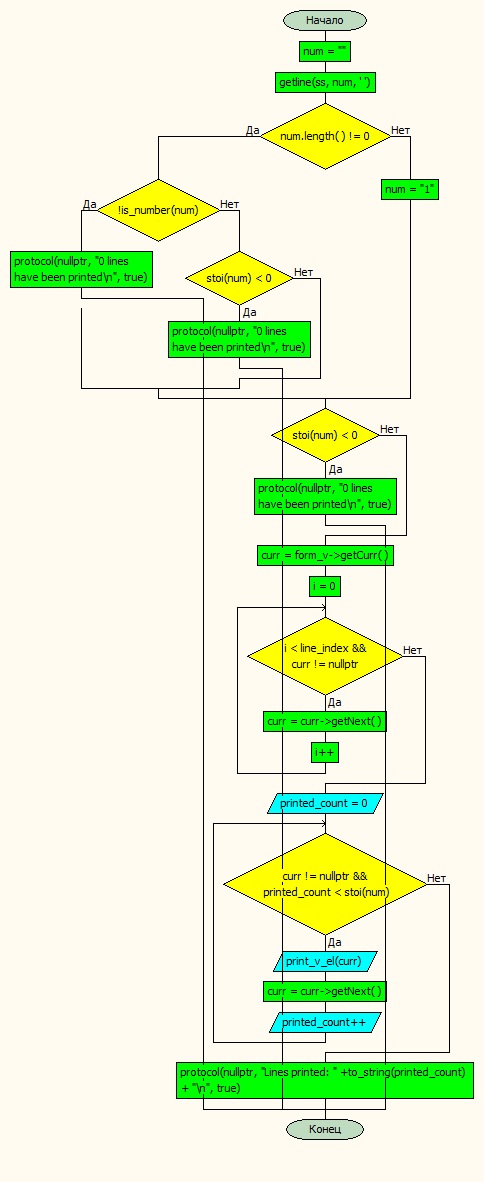
# **Написанные функции:**

|  |  |
| --- | --- |
| read, read\_from\_console | Ввод из файла или из консоли, создание классов |
| protocol | Ведение протокола |
| set\_line\_index | Установка текущей строки |
| print | Вывод строк в консоль |
| del | Удаление |
| ins | Вставка |
| replace | Замена |
| write | Запись в файл |
| out | Вывод в файл |
| clear\_file | Удаление содержимого файла или создание нового |

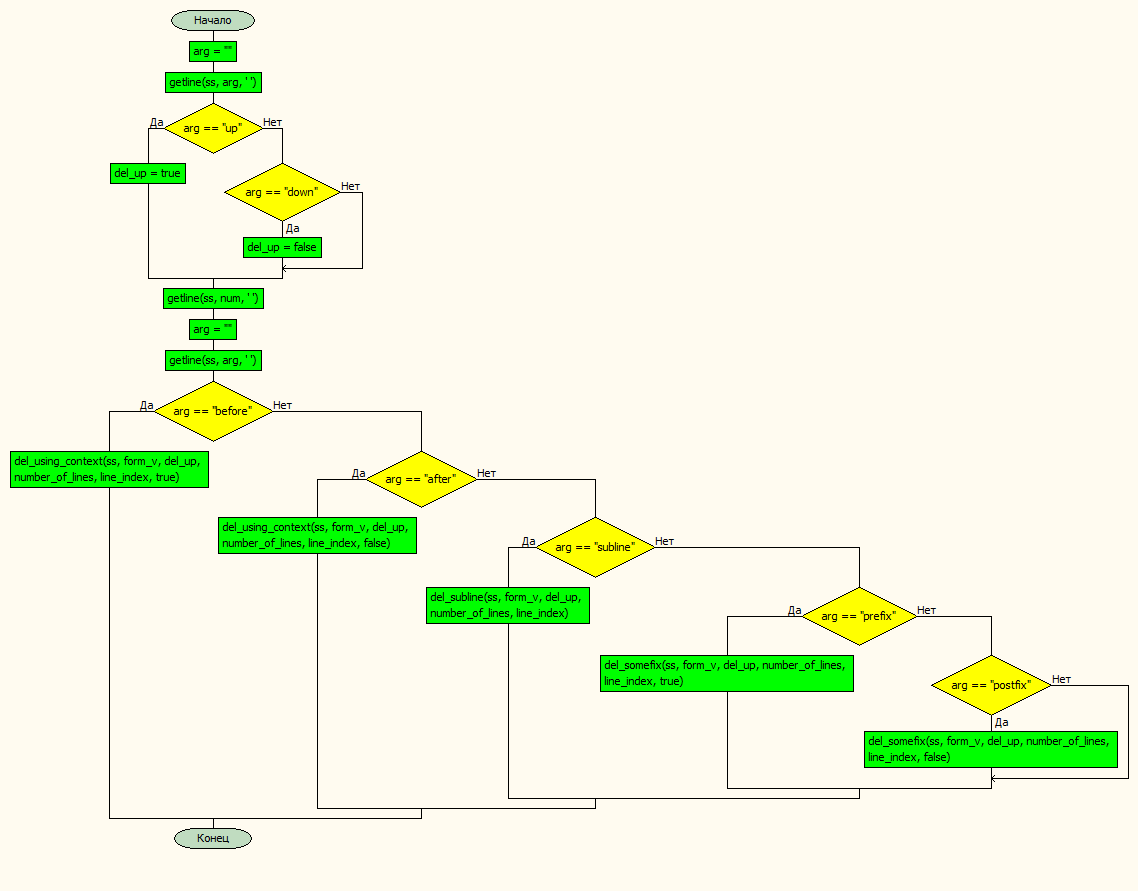
**main:**

****

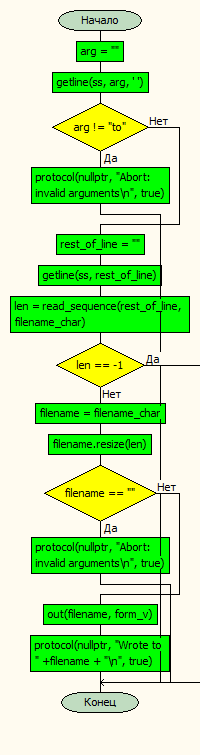
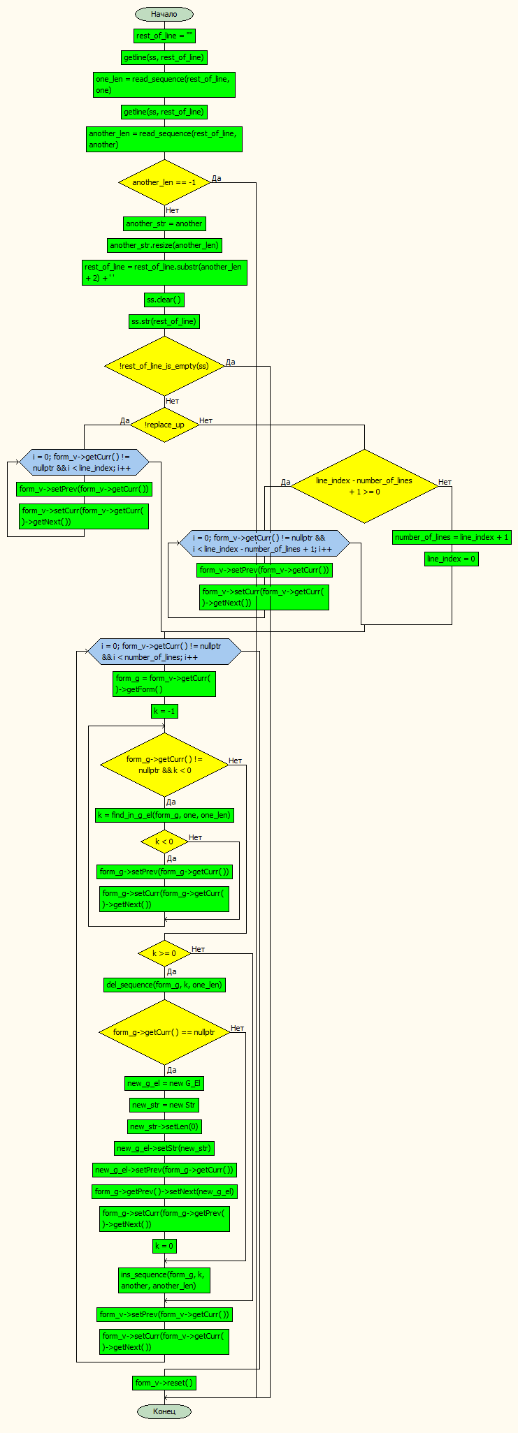
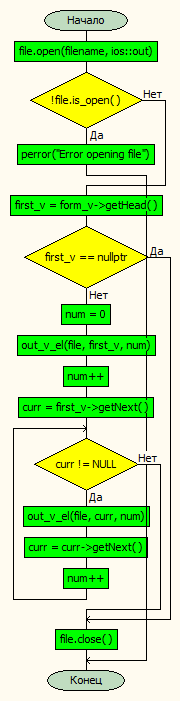
**set\_line\_index, print:**

** **

**del:**

****

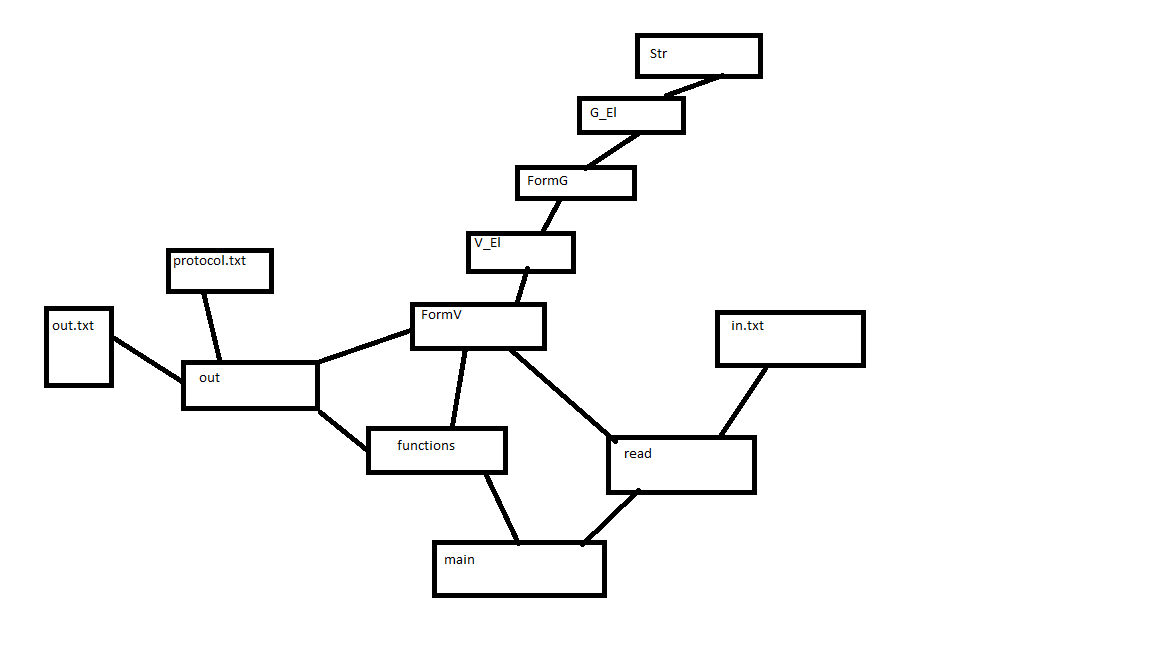
**replace, write, out:**

** **

# 

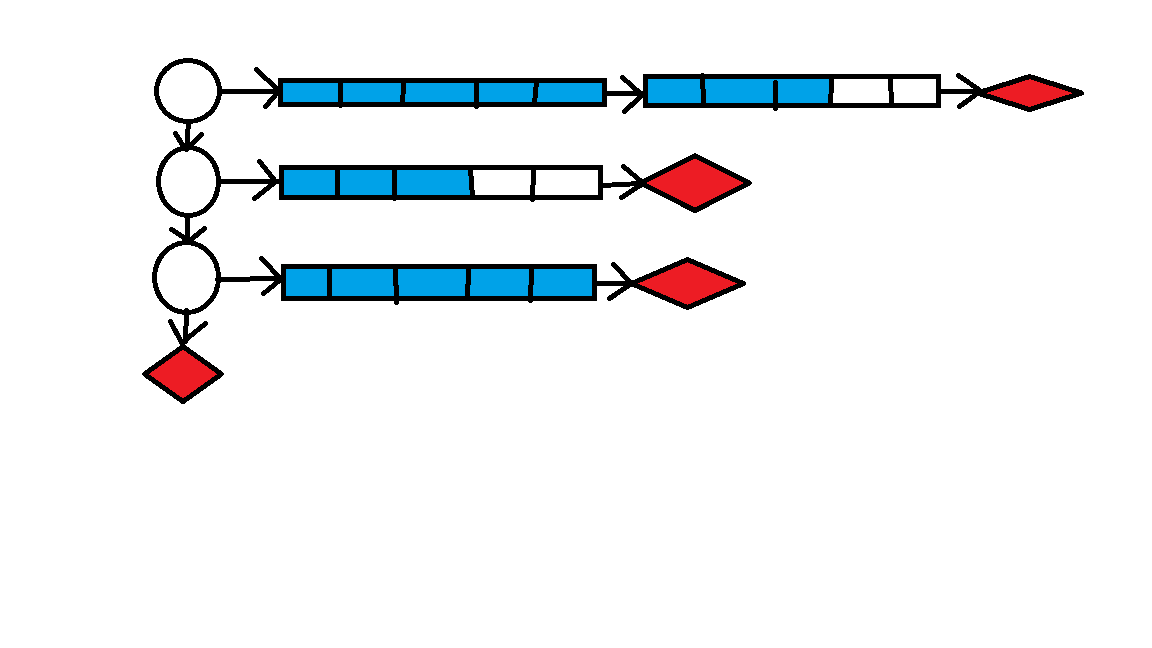
# **Разбиение программы на файлы**

|  |  |
| --- | --- |
| main.cpp | Основа программы |
| read.h, read.cpp | Чтение из файла или из консоли |
| out.h, out.cpp | Вывод в файл |
| functions.h, functions.cpp | Функции обработки |
| FormV.h, FormV.cpp | Вертикальный формуляр |
| V\_El.h, V\_El.cpp | Вертикальный элемент |
| FormG.h, FormG.cpp | Горизонтальный формуляр |
| G\_El.h, G\_El.cpp | Горизонтальный элемент |
| Str.h, Str.cpp | Строка |
| in.txt | Ввод |
| out.txt | Вывод |
| protocol.txt | Протокол |



# Алгоритмы и структуры данных

**Список блоков символов:**



# Текст программы

Main.cpp

#include "functions.h"

#include "read.h"

// max\_line\_len from console DONE

// read from file DONE

// read from console DONE

// set current line(index of line) DONE

// exit DONE

// line DONE

// print DONE

// delete DONE

// insert DONE

// replace DONE

// write to file DONE

int main()

{

cout << "Author: Novikov G.V.\n"

"Start date: 06.06.2022\n"

"End date: 16.09.2022\n"

"Version: cursed.01\n" << endl;

clear\_file(PROTOCOL\_FILENAME);

cout << "Enter max line size: ";

int LINE\_SIZE = 0;

cin >> LINE\_SIZE;

if (LINE\_SIZE <= 0) {

cerr << "Error: invalid line size" << endl;

return 0;

}

cout << "Enter input mode('f' for file, 'c' for console): ";

string input\_mode = "";

cin >> input\_mode;

FormV\* form\_v;

if (input\_mode == "f") {

string filename = "in.txt";

form\_v = read(filename, LINE\_SIZE);

}

else if (input\_mode == "c") {

cout << "Enter number of lines: ";

int n = -1;

cin >> n;

if (n < 0) {

cerr << "Error: number of lines cannot be < 0" << endl;

return 0;

}

form\_v = read\_from\_console(n, LINE\_SIZE);

cout << ">>";

}

else {

cerr << "Error: invalid input mode" << endl;

return 0;

}

string protocol\_str = "Max line size: " + to\_string(LINE\_SIZE) + "\n";

protocol\_str = protocol\_str + "Input mode: " + input\_mode + "\n\n";

protocol\_str = protocol\_str + "INPUT";

protocol(form\_v, protocol\_str, false);

string command = "";

string line = "";

int line\_index = 0;

bool success = true;

while (true) {

success = true;

getline(cin, line);

stringstream ss(line);

getline(ss, command, ' ');

if (command != "") protocol(form\_v, "Command '" + command + "'", true);

if (command == "exit") {

// exit - exit

protocol(nullptr, "Exit program", true);

break;

}

else if (command == "line") {

// line - print current line number

// line <int> - go to line

set\_line\_index(ss, line\_index);

}

else if (command == "print") {

// print <int> - print given number of lines

print(ss, form\_v, line\_index);

}

else if (command == "del") {

// del down... - in lines starting with current 1

// del up... - in lines finishing with current 1

// del...after <string>... - after subtext |-------------->| 2

// del...before <string>... - before subtext |-------------->| 2

// del...subline "<string>" - subline | 2

// del...prefix <int> - first N symbols of line |------>| 2

// del...postfix <int> - last N symbols of line |------>| 2

// del...<int> symbols... - number of symbols |<--------------| 3

del(ss, form\_v, line\_index);

}

else if (command == "ins") {

// ins...after <string>... - after subtext 1

// ins...before <string>... - before subtext 1

// ins...subline "<string>" - subline 2

ins(ss, form\_v, line\_index);

}

else if (command == "replace") {

// replace <string> with <string>

replace(ss, form\_v, line\_index);

}

else if (command == "write") {

// write to <filename>

write(ss, form\_v);

}

else if (command != "") {

cout << "Unknown command: " << command << endl;

protocol(nullptr, "Unknown command\n", true);

}

cout << ">>";

}

desintegrate(form\_v);

return 0;

}

Read.cpp

#include "read.h"

FormV\* read(string filename, unsigned int max\_line\_len) {

fstream file;

file.open(filename, ios::in);

if (!file.is\_open()) {

perror("Error opening file");

exit(1);

}

FormV\* form\_v = new FormV;

V\_El\* first = new V\_El;

string line;

getline(file, line);

FormG\* form\_g = create\_formG((char\*)line.c\_str(), max\_line\_len);

first->setForm(form\_g);

V\_El\* prev;

V\_El\* curr = new V\_El;

curr = first;

while (getline(file, line)) {

prev = curr;

curr = new V\_El;

form\_g = create\_formG((char\*)line.c\_str(), max\_line\_len);

prev->setNext(curr);

curr->setForm(form\_g);

}

form\_v->setHead(first);

form\_v->setCurr(first);

file.close();

return form\_v;

}

FormV\* read\_from\_console(int n, unsigned int max\_line\_len) {

if (n <= 0) {

return nullptr;

}

cout << "ENTER LINES" << endl;

FormV\* form\_v = new FormV;

V\_El\* first = new V\_El;

string line;

cin.ignore();

getline(cin, line);

FormG\* form\_g = create\_formG((char\*)line.c\_str(), max\_line\_len);

first->setForm(form\_g);

V\_El\* prev;

V\_El\* curr = new V\_El;

curr = first;

for (int i = 1; i < n; i++) {

prev = curr;

curr = new V\_El;

getline(cin, line);

form\_g = create\_formG((char\*)line.c\_str(), max\_line\_len);

prev->setNext(curr);

curr->setForm(form\_g);

}

form\_v->setHead(first);

form\_v->setCurr(first);

return form\_v;

}

FormG\* create\_formG(char\* line, unsigned int max\_line\_len) {

FormG\* form = new FormG;

if (max\_line\_len < MAX\_STR\_LEN) {

G\_El\* head = create\_G\_El(line, 0, max\_line\_len);

form->setHead(head);

form->setCurr(head);

return form;

}

G\_El\* head = create\_G\_El(line, 0, MAX\_STR\_LEN);

G\_El\* curr = new G\_El;

curr = head;

G\_El\* prev = new G\_El;

unsigned int line\_len = 0;

while (curr->getStr()->getLen() == MAX\_STR\_LEN && line\_len < max\_line\_len) {

if (line\_len + MAX\_STR\_LEN > max\_line\_len) {

line\_len = max\_line\_len;

}

else {

line\_len += MAX\_STR\_LEN;

}

prev = curr;

curr = create\_G\_El(line, line\_len, max\_line\_len - line\_len);

if (curr->getStr()->getLen() != 0) {

prev->setNext(curr);

curr->setPrev(prev);

}

}

form->setCurr(head);

form->setHead(head);

return form;

}

G\_El\* create\_G\_El(char\* line, unsigned int i, unsigned int str\_len) {

G\_El\* el = new G\_El;

Str\* str = create\_str(line, i, str\_len);

el->setStr(str);

return el;

}

Str\* create\_str(char\* line, unsigned int i, unsigned int str\_len) {

Str\* str = new Str;

unsigned int len = 0;

char sym = line[i];

while (sym != '\0' && len < str\_len && len < MAX\_STR\_LEN) {

str->setLetter(len, sym);

len++;

i++;

sym = line[i];

}

if (len < MAX\_STR\_LEN) {

str->setLen(len);

}

return str;

}

Read.h

#pragma once

#include "FormV.h"

FormV\* read(string filename, unsigned int max\_line\_len);

FormV\* read\_from\_console(int n, unsigned int max\_line\_len);

FormG\* create\_formG(char\* line, unsigned int max\_line\_len);

G\_El\* create\_G\_El(char\* line, unsigned int i, unsigned int str\_len);

Str\* create\_str(char\* line, unsigned int i, unsigned int str\_len);

Out.cpp

#include "out.h"

void log\_g\_el(fstream& file, G\_El\* g\_el) {

file << '"';

for (int i = 0; i < g\_el->getStr()->getLen(); i++) {

file << g\_el->getStr()->getLetter(i);

}

file << '"';

}

void log\_v\_el(fstream& file, V\_El\* v\_el, int num) {

if (num != 0) {

file << "-> ";

}

file << num << ". ";

G\_El\* first\_g = v\_el->getForm()->getHead();

log\_g\_el(file, first\_g);

G\_El\* curr = first\_g->getNext();

while (curr != NULL) {

file << "->";

log\_g\_el(file, curr);

curr = curr->getNext();

}

file << "->NULL" << endl;

}

void clear\_file(string filename) {

fstream file;

file.open(filename, ios::out);

if (!file.is\_open()) {

perror("Error opening file");

exit(1);

}

file.close();

}

void protocol(FormV\* form\_v, string text, bool only\_text) {

fstream file;

file.open(PROTOCOL\_FILENAME, ios::app);

if (!file.is\_open()) {

perror("Error opening file");

exit(1);

}

file << text << endl;

if (!only\_text) {

V\_El\* first\_v = form\_v->getHead();

if (first\_v == nullptr) {

file << "The list is empty" << endl << endl;

}

else {

int num = 0;

log\_v\_el(file, first\_v, num);

num++;

V\_El\* curr = first\_v->getNext();

while (curr != NULL) {

log\_v\_el(file, curr, num);

curr = curr->getNext();

num++;

}

file << "-> NULL" << endl << endl;

}

}

file.close();

}

void out\_g\_el(fstream& file, G\_El\* g\_el) {

for (int i = 0; i < g\_el->getStr()->getLen(); i++) {

file << g\_el->getStr()->getLetter(i);

}

}

void out\_v\_el(fstream& file, V\_El\* v\_el, int num) {

G\_El\* first\_g = v\_el->getForm()->getHead();

out\_g\_el(file, first\_g);

G\_El\* curr = first\_g->getNext();

while (curr != NULL) {

out\_g\_el(file, curr);

curr = curr->getNext();

}

}

void out(string filename, FormV\* form\_v) {

fstream file;

file.open(filename, ios::out);

if (!file.is\_open()) {

perror("Error opening file");

cerr << filename << endl;

return;

}

V\_El\* first\_v = form\_v->getHead();

if (first\_v == nullptr) {

file << "The list is empty" << endl << endl;

}

else {

int num = 0;

out\_v\_el(file, first\_v, num);

num++;

V\_El\* curr = first\_v->getNext();

while (curr != NULL) {

file << endl;

out\_v\_el(file, curr, num);

curr = curr->getNext();

num++;

}

}

file.close();

}

Out.h

#pragma once

#include "FormV.h"

void out(string filename, FormV\* form\_v);

void clear\_file(string filename);

void protocol(FormV\* form\_v, string text, bool only\_text);

//void error()

Functions.cpp

#include "functions.h"

Str\* copy\_str(Str\* str) {

Str\* str\_copy = new Str;

for (int i = 0; i < str->getLen(); i++) {

str\_copy->setLetter(i, str->getLetter(i));

}

str\_copy->setLen(str->getLen());

return str\_copy;

}

FormG\* copy\_form\_g(FormG\* form\_g) {

FormG\* form\_g\_copy = new FormG;

G\_El\* curr\_g\_el = form\_g->getCurr();

G\_El\* prev\_g\_el = form\_g->getPrev();

G\_El\* curr\_g\_el\_copy = nullptr;

G\_El\* prev\_g\_el\_copy = nullptr;

while (form\_g->getCurr() != nullptr) {

G\_El\* g\_el = form\_g->getCurr();

G\_El\* g\_el\_copy = new G\_El;

Str\* str = g\_el->getStr();

Str\* str\_copy = copy\_str(str);

g\_el\_copy->setStr(str\_copy);

if (form\_g\_copy->getCurr() == nullptr) {

form\_g\_copy->setHead(g\_el\_copy);

form\_g\_copy->setCurr(g\_el\_copy);

}

else {

form\_g\_copy->setPrev(form\_g\_copy->getCurr());

form\_g\_copy->setCurr(g\_el\_copy);

form\_g\_copy->getPrev()->setNext(g\_el\_copy);

form\_g\_copy->getCurr()->setPrev(form\_g\_copy->getPrev());

}

if (form\_g\_copy->getCurr()->equals(curr\_g\_el)) {

curr\_g\_el\_copy = form\_g\_copy->getCurr();

prev\_g\_el\_copy = form\_g\_copy->getPrev();

}

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

form\_g->setCurr(curr\_g\_el);

form\_g->setPrev(prev\_g\_el);

form\_g\_copy->setCurr(curr\_g\_el\_copy);

form\_g\_copy->setPrev(prev\_g\_el\_copy);

return form\_g\_copy;

}

//FormV\* copy(FormV\* form\_v) {

// FormV\* form\_v\_copy = new FormV;

// V\_El\* v\_el\_copy\_prev = nullptr;

//

// while (form\_v->getCurr() != nullptr) {

// V\_El\* v\_el = form\_v->getCurr();

// V\_El\* v\_el\_copy = new V\_El;

// FormG\* form\_g = v\_el->getForm();

// FormG\* form\_g\_copy = copy\_form\_g(form\_g);

// v\_el\_copy->setForm(form\_g\_copy);

//

// if (v\_el\_copy\_prev == nullptr) {

// form\_v\_copy->setHead(v\_el\_copy);

// }

// else {

// v\_el\_copy\_prev->setNext(v\_el\_copy);

// }

// v\_el\_copy\_prev = v\_el\_copy;

// form\_v->setPrev(form\_v->getCurr());

// form\_v->setCurr(form\_v->getCurr()->getNext());

// }

//

// form\_v->setCurr(form\_v->getHead());

// form\_v->setPrev(nullptr);

// form\_v\_copy->setCurr(form\_v\_copy->getHead());

// form\_v\_copy->setPrev(nullptr);

//

// return form\_v\_copy;

//}

//

//

//bool el\_is\_in(G\_El\* target\_el, FormV\* form\_v) {

// while (form\_v->getCurr() != nullptr) {

// V\_El\* curr\_v\_el = form\_v->getCurr();

// FormG\* form\_g = curr\_v\_el->getForm();

// while (form\_g->getCurr() != nullptr) {

// if (target\_el->equals(form\_g->getCurr())) {

// form\_g->setCurr(form\_g->getHead());

// form\_v->setCurr(form\_v->getHead());

// return true;

// }

// form\_g->setCurr(form\_g->getCurr()->getNext());

// }

// form\_g->setCurr(form\_g->getHead());

// form\_v->setCurr(form\_v->getCurr()->getNext());

// }

// form\_v->setCurr(form\_v->getHead());

// return false;

//}

//

//

//void handle\_line(V\_El\* curr\_v\_el, FormV\* form\_v\_2) {

// FormG\* form\_g\_1 = curr\_v\_el->getForm();

// while (form\_g\_1->getCurr() != nullptr) {

// if (el\_is\_in(form\_g\_1->getCurr(), form\_v\_2)) {

// if (form\_g\_1->getCurr() == form\_g\_1->getHead()) {

// form\_g\_1->setHead(form\_g\_1->getCurr()->getNext());

// delete form\_g\_1->getCurr();

// form\_g\_1->setCurr(form\_g\_1->getHead());

// form\_g\_1->setPrev(nullptr);

// }

// else {

// form\_g\_1->getPrev()->setNext(form\_g\_1->getCurr()->getNext());

// delete form\_g\_1->getCurr();

// form\_g\_1->setCurr(form\_g\_1->getPrev()->getNext());

// }

// }

// else {

// form\_g\_1->setPrev(form\_g\_1->getCurr());

// form\_g\_1->setCurr(form\_g\_1->getCurr()->getNext());

// }

// }

// form\_g\_1->setCurr(form\_g\_1->getHead());

// form\_g\_1->setPrev(nullptr);

//}

//

//

//void handle\_text(FormV\* form\_v\_1, FormV\* form\_v\_2) {

// while (form\_v\_1->getCurr() != nullptr) {

// handle\_line(form\_v\_1->getCurr(), form\_v\_2);

//

// if (form\_v\_1->getCurr()->getForm()->getHead() == nullptr) {

// if (form\_v\_1->getCurr() == form\_v\_1->getHead()) {

// form\_v\_1->setHead(form\_v\_1->getCurr()->getNext());

// delete form\_v\_1->getCurr();

// form\_v\_1->setCurr(form\_v\_1->getHead());

// form\_v\_1->setPrev(nullptr);

// }

// else {

// form\_v\_1->getPrev()->setNext(form\_v\_1->getCurr()->getNext());

// delete form\_v\_1->getCurr();

// form\_v\_1->setCurr(form\_v\_1->getPrev()->getNext());

// }

// }

// else {

// form\_v\_1->setPrev(form\_v\_1->getCurr());

// form\_v\_1->setCurr(form\_v\_1->getCurr()->getNext());

// }

// }

//

// form\_v\_1->setCurr(form\_v\_1->getHead());

// form\_v\_1->setPrev(nullptr);

//}

//

//

//FormV\* subtract(FormV\* divided, FormV\* dividor) {

// FormV\* result = copy(divided);

// handle\_text(result, dividor);

// return result;

//}

//

//

//bool belongs(FormV\* small\_set, FormV\* large\_set) {

// FormV\* diff = subtract(small\_set, large\_set);

// if (diff->getHead() == nullptr) {

// return true;

// }

// return false;

//}

void desintegrate(FormV\* form\_v) {

while (form\_v->getCurr() != nullptr) {

V\_El\* v\_el = form\_v->getCurr();

FormG\* form\_g = v\_el->getForm();

while (form\_g->getCurr() != nullptr) {

delete form\_g->getCurr()->getStr();

form\_g->setCurr(form\_g->getCurr()->getNext());

delete form\_g->getPrev();

}

form\_v->setCurr(form\_v->getCurr()->getNext());

delete form\_v->getPrev();

}

delete form\_v;

}

void set\_line\_index(stringstream& ss, int& line\_index) {

bool success = true;

string index = "";

getline(ss, index, ' ');

if (index.length() != 0) {

for (int i = 0; i < index.length(); i++) {

if (!isdigit(index[i])) {

if (i == 0 && index[i] == '-') {

cerr << "Error: current line number must be greater than 0" << endl;

}

else {

cerr << "Error: current line number must be an integer" << endl;

}

cout << "Current line number is " << line\_index + 1 << endl;

success = false;

}

}

}

else {

cout << "Current line number is " << line\_index + 1 << endl;

success = false;

}

if (success && stoi(index) == 0) {

cerr << "Error: current line number must be greater than 0" << endl;

cout << "Current line number is " << line\_index + 1 << endl;

success = false;

}

if (success) {

line\_index = stoi(index) - 1;

cout << "Line number has been set to " << index << endl;

protocol(nullptr, "Line number has been set to " + index + "\n", true);

}

else {

protocol(nullptr, "Line number has not been set\n", true);

}

}

void print\_g\_el(G\_El\* g\_el) {

for (int i = 0; i < g\_el->getStr()->getLen(); i++) {

cout << g\_el->getStr()->getLetter(i);

}

}

void print\_v\_el(V\_El\* v\_el) {

G\_El\* first\_g = v\_el->getForm()->getHead();

print\_g\_el(first\_g);

G\_El\* curr = first\_g->getNext();

while (curr != NULL) {

print\_g\_el(curr);

curr = curr->getNext();

}

cout << endl;

}

bool is\_number(string num) {

if (num.length() == 0) return false;

for (int i = 0; i < num.length(); i++) {

if (!isdigit(num[i]) && !(i == 0 && num[i] == '-')) {

return false;

}

}

return true;

}

void print(stringstream& ss, FormV\* form\_v, int line\_index) {

string num = "";

getline(ss, num, ' ');

if (num.length() != 0) {

if (!is\_number(num)) {

cerr << "Error: number of lines must be an integer" << endl;

protocol(nullptr, "0 lines have been printed\n", true);

return;

}

else if (stoi(num) < 0) {

cerr << "Error: number of lines must be 0 or greater" << endl;

protocol(nullptr, "0 lines have been printed\n", true);

return;

}

}

else {

num = "1";

}

if (stoi(num) < 0) {

cerr << "Error: number of lines must be 0 or greater" << endl;

protocol(nullptr, "0 lines have been printed\n", true);

return;

}

V\_El\* curr = form\_v->getCurr();

int i = 0;

while (i < line\_index && curr != nullptr) {

curr = curr->getNext();

i++;

}

int printed\_count = 0;

while (curr != nullptr && printed\_count < stoi(num)) {

print\_v\_el(curr);

curr = curr->getNext();

printed\_count++;

}

protocol(nullptr, "Lines printed: " + to\_string(printed\_count) + "\n", true);

}

void throw\_arg\_exception(string arg, int len = -1) {

cerr << "Error: invalid argument: '";

if (len < 0) {

cout << arg;

}

else {

for (int i = 0; i < len; i++) {

cout << arg[i];

}

}

cout << "'" << endl;

}

int read\_sequence(string rest\_of\_line, char\* sequence) {

if (rest\_of\_line == "") {

cerr << "Error: sequence is not provided" << endl;

return -1;

}

int i = 0;

while (i < rest\_of\_line.length() && rest\_of\_line[i] == ' ') i++;

if (rest\_of\_line[i] != '"') {

cerr << "Error: sequence must be inside double quotes" << endl;

return -1;

}

i++;

int j = 0;

for (; i + j < rest\_of\_line.length(); j++) {

if (rest\_of\_line[i + j] == '"') break;

sequence[j] = rest\_of\_line[i + j];

}

if (rest\_of\_line[i + j] != '"') {

cerr << "Error: sequence must be inside double quotes" << endl;

return -1;

}

if (i + j + 1 < rest\_of\_line.length() && rest\_of\_line[i + j + 1] != ' ') {

int k = i + j;

for (; (k < rest\_of\_line.length() && rest\_of\_line[k] != ' '); k++) {

sequence[k - i] = rest\_of\_line[k];

}

throw\_arg\_exception('"' + string(sequence), k - i + 1);

return -1;

}

if (j == 0) {

cerr << "Error: sequence is not provided" << endl;

return -1;

}

return j;

}

int read\_number\_of\_symbols(stringstream& ss) {

string num = "";

getline(ss, num, ' ');

getline(ss, num, ' ');

string symbols\_word = "";

getline(ss, symbols\_word, ' ');

if (num.length() == 0) {

cerr << "Error: number of symbols is not provided" << endl;

return -1;

}

if (!is\_number(num)) {

throw\_arg\_exception(num);

return -1;

}

if (symbols\_word != "symbols" && symbols\_word != "") {

throw\_arg\_exception(symbols\_word);

return -1;

}

if (stoi(num) < 0) {

cerr << "Error: number of symbols must be 0 or greater" << endl;

return -1;

}

return stoi(num);

}

bool rest\_of\_line\_is\_empty(stringstream& ss) {

string arg = "";

getline(ss, arg, ' ');

getline(ss, arg, ' ');

if (arg != "") {

throw\_arg\_exception(arg);

return false;

}

return true;

}

void del\_line\_if\_empty(FormV\* form\_v) {

if (form\_v->getCurr()->getForm()->getHead() == nullptr) {

if (form\_v->getCurr() == form\_v->getHead()) {

form\_v->setHead(form\_v->getCurr()->getNext());

delete form\_v->getCurr();

form\_v->setCurr(form\_v->getHead());

}

else {

form\_v->getPrev()->setNext(form\_v->getCurr()->getNext());

delete form\_v->getCurr();

form\_v->setCurr(form\_v->getPrev()->getNext());

}

}

else {

form\_v->setPrev(form\_v->getCurr());

if (form\_v->getCurr() != nullptr) form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

int find\_in\_g\_el(FormG\* form\_g, char\* subline, int len) {

FormG\* form\_g\_copy = copy\_form\_g(form\_g);

Str\* str = form\_g->getCurr()->getStr();

int init\_str\_len = str->getLen();

for (int i = 0; i < init\_str\_len; i++) {

form\_g\_copy = copy\_form\_g(form\_g);

str = form\_g->getCurr()->getStr();

int j = 0;

if (i + len <= str->getLen()) {

while (j < len && str->getLetter(i + j) == subline[j]) j++;

}

else {

while (j + i < str->getLen() && str->getLetter(i + j) == subline[j]) j++;

if (j > 0) {

while (form\_g\_copy->getCurr()->getNext() != nullptr && j < len) {

int j\_in\_str = 0;

form\_g\_copy->setCurr(form\_g\_copy->getCurr()->getNext());

str = form\_g\_copy->getCurr()->getStr();

while (j\_in\_str < str->getLen() && j < len && str->getLetter(j\_in\_str) == subline[j]) {

j\_in\_str++;

j++;

}

if (j < len && j\_in\_str < str->getLen()) break;

}

}

}

if (j == len) {

return i;

}

}

return -1;

}

void del\_sequence(FormG\* form\_g, int i, int len) {

// deletion place is identified by form\_g->getCurr() for g\_el and i for index in g\_el

if (form\_g->getCurr() == nullptr) return;

// first el

Str\* str = form\_g->getCurr()->getStr();

if (i + len <= str->getLen()) {

for (int j = i; j + len < str->getLen(); j++) {

str->setLetter(j, str->getLetter(j + len));

}

str->setLen(str->getLen() - len);

if (str->getLen() == 0) {

if (form\_g->getPrev() == nullptr && form\_g->getCurr()->getNext() == nullptr) {

delete form\_g->getCurr();

form\_g->setCurr(nullptr);

form\_g->setHead(nullptr);

}

else if (form\_g->getPrev() == nullptr) {

form\_g->setCurr(form\_g->getCurr()->getNext());

form\_g->setHead(form\_g->getCurr());

delete form\_g->getCurr()->getPrev();

}

else {

form\_g->getPrev()->setNext(form\_g->getCurr()->getNext());

delete form\_g->getCurr();

form\_g->setCurr(form\_g->getPrev()->getNext());

}

delete str;

}

len = 0;

}

else if (i != 0) {

len -= str->getLen();

len += i;

str->setLen(i);

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

while (len > 0 && form\_g->getCurr() != nullptr) {

str = form\_g->getCurr()->getStr();

if (len >= str->getLen()) {

if (form\_g->getPrev() == nullptr && form\_g->getCurr()->getNext() == nullptr) {

delete form\_g->getCurr();

form\_g->setCurr(nullptr);

form\_g->setHead(nullptr);

}

else if (form\_g->getPrev() == nullptr) {

form\_g->setCurr(form\_g->getCurr()->getNext());

form\_g->setHead(form\_g->getCurr());

delete form\_g->getCurr()->getPrev();

}

else {

form\_g->getPrev()->setNext(form\_g->getCurr()->getNext());

delete form\_g->getCurr();

form\_g->setCurr(form\_g->getPrev()->getNext());

}

len -= str->getLen();

delete str;

}

else {

for (int j = 0; j + len < str->getLen(); j++) {

str->setLetter(j, str->getLetter(j + len));

}

str->setLen(str->getLen() - len);

len = 0;

}

}

}

void del\_using\_context(stringstream& ss, FormV\* form\_v, bool del\_up, int number\_of\_lines, int line\_index, bool del\_before) {

protocol(nullptr, "Before context: <bool> " + to\_string(del\_before), true);

string rest\_of\_line = "";

getline(ss, rest\_of\_line);

char context[1000];

int len = read\_sequence(rest\_of\_line, context);

if (len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string context\_str = context;

context\_str.resize(len);

protocol(nullptr, "Context: '" + context\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

int number\_of\_symbols = read\_number\_of\_symbols(ss);

protocol(nullptr, "Number of deleted symbols: " + to\_string(number\_of\_symbols), true);

if (!rest\_of\_line\_is\_empty(ss)) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

if (!del\_up) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else if (line\_index - number\_of\_lines + 1 >= 0) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index - number\_of\_lines + 1; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else {

number\_of\_lines = line\_index + 1;

line\_index = 0;

}

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index + number\_of\_lines; i++) {

FormG\* form\_g = form\_v->getCurr()->getForm();

int k = -1;

while (form\_g->getCurr() != nullptr && k < 0) {

k = find\_in\_g\_el(form\_g, context, len);

if (k < 0) {

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

if (k >= 0) { // deletion place is identified by form\_g->getCurr() for g\_el and k for index in g\_el

int del\_index = -1;

if (!del\_before) {

del\_index = k + len;

}

else {

del\_index = k - number\_of\_symbols;

while (del\_index < 0 && form\_g->getCurr() != form\_g->getHead()) {

form\_g->setCurr(form\_g->getPrev());

form\_g->setPrev(form\_g->getCurr()->getPrev());

del\_index += form\_g->getCurr()->getStr()->getLen();

}

if (del\_index < 0 && form\_g->getCurr() == form\_g->getHead()) {

number\_of\_symbols = k;

del\_index = 0;

}

}

del\_sequence(form\_g, del\_index, number\_of\_symbols);

del\_line\_if\_empty(form\_v);

protocol(form\_v, "AFTER DELETION del\_line index " + to\_string(i), false);

}

else {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

form\_v->reset();

}

void del\_subline(stringstream& ss, FormV\* form\_v, bool del\_up, int number\_of\_lines, int line\_index) {

string rest\_of\_line = "";

getline(ss, rest\_of\_line);

char subline[1000];

int len = read\_sequence(rest\_of\_line, subline);

if (len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string subline\_str = subline;

subline\_str.resize(len);

protocol(nullptr, "Context: '" + subline\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

if (!rest\_of\_line\_is\_empty(ss)) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

if (!del\_up) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else if (line\_index - number\_of\_lines + 1 >= 0) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index - number\_of\_lines + 1; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else {

number\_of\_lines = line\_index + 1;

line\_index = 0;

}

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index + number\_of\_lines; i++) {

FormG\* form\_g = form\_v->getCurr()->getForm();

int k = -1;

while (form\_g->getCurr() != nullptr && k < 0) {

k = find\_in\_g\_el(form\_g, subline, len);

if (k < 0) {

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

if (k >= 0) { // deletion place is identified by form\_g->getCurr() for g\_el and i for index in g\_el

del\_sequence(form\_g, k, len);

del\_line\_if\_empty(form\_v);

protocol(form\_v, "AFTER DELETION del\_line index " + to\_string(i), false);

}

else {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

form\_v->reset();

}

void del\_prefix(FormG\* form\_g, int number\_of\_symbols) {

while (form\_g->getHead() != nullptr && number\_of\_symbols > 0) {

Str\* str = form\_g->getCurr()->getStr();

if (number\_of\_symbols >= str->getLen()) {

number\_of\_symbols -= str->getLen();

form\_g->setHead(form\_g->getCurr()->getNext());

delete form\_g->getCurr()->getStr();

delete form\_g->getCurr();

form\_g->setCurr(form\_g->getHead());

}

else {

for (int i = 0; i + number\_of\_symbols < str->getLen(); i++) {

str->setLetter(i, str->getLetter(i + number\_of\_symbols));

}

str->setLen(str->getLen() - number\_of\_symbols);

number\_of\_symbols = 0;

}

}

}

void del\_postfix(FormG\* form\_g, int number\_of\_symbols) {

while (form\_g->getCurr()->getNext() != nullptr) {

form\_g->setCurr(form\_g->getCurr()->getNext());

}

while (form\_g->getCurr() != nullptr && number\_of\_symbols > 0) {

Str\* str = form\_g->getCurr()->getStr();

if (number\_of\_symbols >= str->getLen()) {

number\_of\_symbols -= str->getLen();

if (form\_g->getCurr()->getPrev() != nullptr) {

form\_g->setCurr(form\_g->getCurr()->getPrev());

delete form\_g->getCurr()->getNext()->getStr();

delete form\_g->getCurr()->getNext();

form\_g->getCurr()->setNext(nullptr);

}

else {

delete form\_g->getCurr()->getStr();

delete form\_g->getCurr();

form\_g->setCurr(nullptr);

form\_g->setHead(nullptr);

}

}

else {

str->setLen(str->getLen() - number\_of\_symbols);

number\_of\_symbols = 0;

}

}

}

void del\_somefix(stringstream& ss, FormV\* form\_v, bool del\_up, int number\_of\_lines, int line\_index, bool is\_prefix) {

protocol(nullptr, "Is prefix: <bool> " + to\_string(del\_up), true);

int number\_of\_symbols = read\_number\_of\_symbols(ss);

protocol(nullptr, "Number of deleted symbols: " + to\_string(number\_of\_symbols), true);

if (!rest\_of\_line\_is\_empty(ss)) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

if (!del\_up) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else if (line\_index - number\_of\_lines + 1 >= 0) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index - number\_of\_lines + 1; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else {

number\_of\_lines = line\_index + 1;

line\_index = 0;

}

for (int i = 0; i < number\_of\_lines && form\_v->getCurr() != nullptr; i++) {

if (is\_prefix) {

del\_prefix(form\_v->getCurr()->getForm(), number\_of\_symbols);

}

else {

del\_postfix(form\_v->getCurr()->getForm(), number\_of\_symbols);

}

del\_line\_if\_empty(form\_v);

protocol(form\_v, "AFTER DELETION del\_line index " + to\_string(i), false);

}

form\_v->reset();

}

void del(stringstream& ss, FormV\* form\_v, int line\_index) {

string arg = "";

getline(ss, arg, ' ');

bool del\_up;

if (arg == "up") {

del\_up = true;

}

else if (arg == "down") {

del\_up = false;

}

else if (arg == "") {

cerr << "Error: arguments not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else {

throw\_arg\_exception(arg);

return;

}

string num= "";

getline(ss, num, ' ');

if (num == "") {

cerr << "Error: number of lines is not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else if (!is\_number(num)) {

cerr << "Error: number of lines must be integer" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

int number\_of\_lines = stoi(num);

if (number\_of\_lines < 0) {

cerr << "Error: number of lines must be 0 or greater" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

protocol(nullptr, "Starting with line index: " + to\_string(line\_index), true);

protocol(nullptr, "Number of lines to process: " + to\_string(number\_of\_lines), true);

protocol(nullptr, "Moving up: <bool> " + to\_string(del\_up), true);

arg = "";

getline(ss, arg, ' ');

if (arg == "before") {

del\_using\_context(ss, form\_v, del\_up, number\_of\_lines, line\_index, true);

}

else if (arg == "after") {

del\_using\_context(ss, form\_v, del\_up, number\_of\_lines, line\_index, false);

}

else if (arg == "subline") {

del\_subline(ss, form\_v, del\_up, number\_of\_lines, line\_index);

}

else if (arg == "prefix") {

del\_somefix(ss, form\_v, del\_up, number\_of\_lines, line\_index, true);

}

else if (arg == "postfix") {

del\_somefix(ss, form\_v, del\_up, number\_of\_lines, line\_index, false);

}

else if (arg == "") {

cerr << "Error: arguments not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else {

throw\_arg\_exception(arg);

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

}

void ins\_sequence(FormG\* form\_g, const int i, const char\* seq, int len) {

// inserts before i

Str\* str = form\_g->getCurr()->getStr();

if (len <= MAX\_STR\_LEN - str->getLen()) {

for (int j = str->getLen() - 1; j >= i; j--) {

str->setLetter(j + len, str->getLetter(j));

}

for (int j = 0; j < len; j++) {

str->setLetter(i + j, seq[j]);

}

str->setLen(str->getLen() + len);

}

else {

// move letters in current str into new el

G\_El\* new\_g\_el = new G\_El;

Str\* new\_str = new Str;

unsigned int str\_len = 0;

for (int k = i; k < str->getLen(); k++) {

new\_str->setLetter(str\_len, str->getLetter(k));

str\_len++;

}

new\_str->setLen(str\_len);

new\_g\_el->setStr(new\_str);

new\_g\_el->setPrev(form\_g->getCurr());

new\_g\_el->setNext(form\_g->getCurr()->getNext());

str->setLen(i);

form\_g->getCurr()->setNext(new\_g\_el);

// copy from seq into current el after what remained from str

int j = 0;

for (; i + j < MAX\_STR\_LEN && j < len; j++) {

str->setLetter(i + j, seq[j]);

}

str->setLen(i + j);

// create g\_el and insert it after current, repeat until j == len

while (j < len) {

G\_El\* new\_g\_el = new G\_El;

Str\* new\_str = new Str;

str\_len = 0;

while (j < len && str\_len < MAX\_STR\_LEN) {

new\_str->setLetter(str\_len, seq[j]);

j++;

str\_len++;

}

new\_str->setLen(str\_len);

new\_g\_el->setStr(new\_str);

new\_g\_el->setPrev(form\_g->getCurr());

new\_g\_el->setNext(form\_g->getCurr()->getNext());

form\_g->getCurr()->setNext(new\_g\_el);

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

}

void ins\_using\_context(stringstream& ss, FormV\* form\_v, bool ins\_up, int number\_of\_lines, int line\_index, bool ins\_before) {

protocol(nullptr, "Before context: <bool> " + to\_string(ins\_before), true);

string rest\_of\_line = "";

getline(ss, rest\_of\_line);

char context[1000];

int cont\_len = read\_sequence(rest\_of\_line, context);

if (cont\_len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string context\_str = context;

context\_str.resize(cont\_len);

protocol(nullptr, "Context: '" + context\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(cont\_len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

string arg = "";

getline(ss, arg, ' ');

getline(ss, arg, ' ');

if (arg != "subline") {

throw\_arg\_exception(arg);

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

getline(ss, rest\_of\_line);

char inserted\_seq[1000];

int seq\_len = read\_sequence(rest\_of\_line, inserted\_seq);

if (seq\_len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string inserted\_seq\_str = inserted\_seq;

inserted\_seq\_str.resize(seq\_len);

protocol(nullptr, "Inserted sequence: '" + inserted\_seq\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(seq\_len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

if (!rest\_of\_line\_is\_empty(ss)) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

if (!ins\_up) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else if (line\_index - number\_of\_lines + 1 >= 0) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index - number\_of\_lines + 1; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else {

number\_of\_lines = line\_index + 1;

line\_index = 0;

}

for (int i = 0; form\_v->getCurr() != nullptr && i < number\_of\_lines; i++) {

FormG\* form\_g = form\_v->getCurr()->getForm();

int k = -1;

while (form\_g->getCurr() != nullptr && k < 0) {

k = find\_in\_g\_el(form\_g, context, cont\_len);

if (k < 0) {

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

if (k >= 0) { // insertion place is identified by form\_g->getCurr() for g\_el and k for index in g\_el

int ins\_index = -1;

if (ins\_before) {

ins\_index = k;

}

else {

ins\_index = k + cont\_len;

while (form\_g->getCurr()->getNext() != nullptr && ins\_index >= form\_g->getCurr()->getStr()->getLen()) {

ins\_index -= form\_g->getCurr()->getStr()->getLen();

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

if (form\_g->getCurr()->getNext() == nullptr && ins\_index >= form\_g->getCurr()->getStr()->getLen()) {

G\_El\* new\_g\_el = new G\_El;

Str\* new\_str = new Str;

new\_str->setLen(0);

new\_g\_el->setStr(new\_str);

new\_g\_el->setPrev(form\_g->getCurr());

form\_g->getCurr()->setNext(new\_g\_el);

ins\_index = 0;

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

ins\_sequence(form\_g, ins\_index, inserted\_seq, seq\_len); // inserts before ins\_index

protocol(form\_v, "AFTER INSERTION ins\_line index " + to\_string(i), false);

}

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

form\_v->reset();

}

void ins(stringstream& ss, FormV\* form\_v, int line\_index) {

string arg = "";

getline(ss, arg, ' ');

bool ins\_up;

if (arg == "up") {

ins\_up = true;

}

else if (arg == "down") {

ins\_up = false;

}

else if (arg == "") {

cerr << "Error: arguments not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else {

throw\_arg\_exception(arg);

return;

}

string num= "";

getline(ss, num, ' ');

if (num == "") {

cerr << "Error: number of lines is not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else if (!is\_number(num)) {

cerr << "Error: number of lines must be integer" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

int number\_of\_lines = stoi(num);

if (number\_of\_lines < 0) {

cerr << "Error: number of lines must be 0 or greater" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

protocol(nullptr, "Starting with line index: " + to\_string(line\_index), true);

protocol(nullptr, "Number of lines to process: " + to\_string(number\_of\_lines), true);

protocol(nullptr, "Moving up: <bool> " + to\_string(ins\_up), true);

arg = "";

getline(ss, arg, ' ');

if (arg == "before") {

ins\_using\_context(ss, form\_v, ins\_up, number\_of\_lines, line\_index, true);

}

else if (arg == "after") {

ins\_using\_context(ss, form\_v, ins\_up, number\_of\_lines, line\_index, false);

}

else if (arg == "") {

cerr << "Error: arguments not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else {

throw\_arg\_exception(arg);

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

}

void replace\_one\_with\_another(stringstream& ss, FormV\* form\_v, bool replace\_up, int number\_of\_lines, int line\_index) {

string rest\_of\_line = "";

getline(ss, rest\_of\_line);

char one[1000];

int one\_len = read\_sequence(rest\_of\_line, one);

if (one\_len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string one\_str = one;

one\_str.resize(one\_len);

protocol(nullptr, "One: '" + one\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(one\_len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

string arg = "";

getline(ss, arg, ' ');

getline(ss, arg, ' ');

if (arg != "with") {

throw\_arg\_exception(arg);

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

getline(ss, rest\_of\_line);

char another[1000];

int another\_len = read\_sequence(rest\_of\_line, another);

if (another\_len == -1) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string another\_str = another;

another\_str.resize(another\_len);

protocol(nullptr, "Another: '" + another\_str + "'", true);

rest\_of\_line = rest\_of\_line.substr(another\_len + 2) + ' '; // +2 because of quotes

ss.clear();

ss.str(rest\_of\_line);

if (!rest\_of\_line\_is\_empty(ss)) {

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

if (!replace\_up) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else if (line\_index - number\_of\_lines + 1 >= 0) {

for (int i = 0; form\_v->getCurr() != nullptr && i < line\_index - number\_of\_lines + 1; i++) {

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

}

else {

number\_of\_lines = line\_index + 1;

line\_index = 0;

}

for (int i = 0; form\_v->getCurr() != nullptr && i < number\_of\_lines; i++) {

FormG\* form\_g = form\_v->getCurr()->getForm();

int k = -1;

while (form\_g->getCurr() != nullptr && k < 0) {

k = find\_in\_g\_el(form\_g, one, one\_len);

if (k < 0) {

form\_g->setPrev(form\_g->getCurr());

form\_g->setCurr(form\_g->getCurr()->getNext());

}

}

if (k >= 0) {

del\_sequence(form\_g, k, one\_len);

if (form\_g->getCurr() == nullptr) {

G\_El\* new\_g\_el = new G\_El;

Str\* new\_str = new Str;

new\_str->setLen(0);

new\_g\_el->setStr(new\_str);

new\_g\_el->setPrev(form\_g->getCurr());

form\_g->getPrev()->setNext(new\_g\_el);

form\_g->setCurr(form\_g->getPrev()->getNext());

k = 0;

}

protocol(form\_v, "AFTER DELETION replace\_line index " + to\_string(i), false);

ins\_sequence(form\_g, k, another, another\_len); // inserts before ins\_index

protocol(form\_v, "AFTER INSERTION replace\_line index " + to\_string(i), false);

}

form\_v->setPrev(form\_v->getCurr());

form\_v->setCurr(form\_v->getCurr()->getNext());

}

form\_v->reset();

}

void replace(stringstream& ss, FormV\* form\_v, int line\_index) {

string arg = "";

getline(ss, arg, ' ');

bool replace\_up;

if (arg == "up") {

replace\_up = true;

}

else if (arg == "down") {

replace\_up = false;

}

else if (arg == "") {

cerr << "Error: arguments not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else {

throw\_arg\_exception(arg);

return;

}

string num= "";

getline(ss, num, ' ');

if (num == "") {

cerr << "Error: number of lines is not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

else if (!is\_number(num)) {

cerr << "Error: number of lines must be integer" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

int number\_of\_lines = stoi(num);

if (number\_of\_lines < 0) {

cerr << "Error: number of lines must be 0 or greater" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

protocol(nullptr, "Starting with line index: " + to\_string(line\_index), true);

protocol(nullptr, "Number of lines to process: " + to\_string(number\_of\_lines), true);

protocol(nullptr, "Moving up: <bool> " + to\_string(replace\_up), true);

replace\_one\_with\_another(ss, form\_v, replace\_up, number\_of\_lines, line\_index);

}

void write(stringstream& ss, FormV\* form\_v) {

string arg = "";

getline(ss, arg, ' ');

if (arg != "to") {

cerr << "Error: invalid command" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

string rest\_of\_line = "";

getline(ss, rest\_of\_line);

char filename\_char[1000];

int len = read\_sequence(rest\_of\_line, filename\_char);

if (len == -1) return;

string filename = filename\_char;

filename.resize(len);

if (filename == "") {

cerr << "Error: filename not provided" << endl;

protocol(nullptr, "Abort: invalid arguments\n", true);

return;

}

out(filename, form\_v);

protocol(nullptr, "Wrote to " + filename + "\n", true);

}

Function.h

#pragma once

#include "FormV.h"

#include "out.h"

//bool belongs(FormV\* small\_set, FormV\* large\_set);

//FormV\* subtract(FormV\* form\_v\_1, FormV\* form\_v\_2);

//bool el\_is\_in(G\_El\* target\_el, FormV\* form\_v);

void desintegrate(FormV\* form\_v);

void set\_line\_index(stringstream& ss, int& line\_index);

void print(stringstream& ss, FormV\* form\_v, int line\_index);

void print\_g\_el(G\_El\* g\_el);

void print\_v\_el(V\_El\* v\_el);

void del\_sequence(FormG\* form\_g, int i, int len);

void del(stringstream& ss, FormV\* form\_v, int line\_index);

void ins(stringstream& ss, FormV\* form\_v, int line\_index);

void replace(stringstream& ss, FormV\* form\_v, int line\_index);

void write(stringstream& ss, FormV\* form\_v);

FormV.cpp

#include "FormV.h"

V\_El\* FormV::getHead() {

return head;

};

void FormV::setHead(V\_El\* new\_head) {

head = new\_head;

};

V\_El\* FormV::getCurr() {

return curr;

};

void FormV::setCurr(V\_El\* new\_curr) {

curr = new\_curr;

};

V\_El\* FormV::getPrev() {

return prev;

};

void FormV::setPrev(V\_El\* new\_prev) {

prev = new\_prev;

};

void FormV::reset() {

curr = head;

while (curr != nullptr) {

curr->getForm()->reset();

curr = curr->getNext();

}

curr = head;

prev = nullptr;

}

FormV.h

#pragma once

#include "V\_El.h"

class FormV {

private:

V\_El\* head = nullptr;

V\_El\* curr = nullptr;

V\_El\* prev = nullptr;

public:

V\_El\* getHead();

void setHead(V\_El\* new\_head);

V\_El\* getCurr();

void setCurr(V\_El\* new\_curr);

V\_El\* getPrev();

void setPrev(V\_El\* new\_prev);

void reset();

};

V\_El.cpp

#include "V\_El.h"

FormG\* V\_El::getForm() {

return form;

};

void V\_El::setForm(FormG\* new\_form) {

form = new\_form;

};

V\_El\* V\_El::getNext() {

return next;

};

void V\_El::setNext(V\_El\* new\_next) {

next = new\_next;

};

V\_El.h

#pragma once

#include "FormG.h"

class V\_El{

private:

FormG\* form = nullptr;

V\_El\* next = nullptr;

public:

FormG\* getForm();

void setForm(FormG\* new\_form);

V\_El\* getNext();

void setNext(V\_El\* new\_next);

};

FormG.cpp

#include "FormG.h"

G\_El\* FormG::getHead() {

return head;

};

void FormG::setHead(G\_El\* new\_head) {

head = new\_head;

};

G\_El\* FormG::getCurr() {

return curr;

};

void FormG::setCurr(G\_El\* new\_curr) {

curr = new\_curr;

};

G\_El\* FormG::getPrev() {

return prev;

};

void FormG::setPrev(G\_El\* new\_prev) {

prev = new\_prev;

};

void FormG::reset() {

curr = head;

prev = nullptr;

}

FormG.h

#pragma once

#include "G\_El.h"

class FormG {

private:

G\_El\* head = nullptr;

G\_El\* curr = nullptr;

G\_El\* prev = nullptr;

public:

G\_El\* getHead();

void setHead(G\_El\* new\_head);

G\_El\* getCurr();

void setCurr(G\_El\* new\_curr);

G\_El\* getPrev();

void setPrev(G\_El\* new\_prev);

void reset();

};

G\_El.cpp

#include "G\_El.h"

G\_El\* G\_El::getNext() {

return next;

};

void G\_El::setNext(G\_El\* new\_next) {

next = new\_next;

};

G\_El\* G\_El::getPrev() {

return prev;

};

void G\_El::setPrev(G\_El\* new\_prev) {

prev = new\_prev;

};

Str\* G\_El::getStr() {

return str;

};

void G\_El::setStr(Str\* new\_str) {

str = new\_str;

};

bool G\_El::equals(G\_El\* el) {

if (this == nullptr && el == nullptr) return true;

if (this == nullptr ^ el == nullptr) return false;

return getStr()->equals(el->getStr());

}

G\_El.h

#pragma once

#include "Str.h"

class G\_El {

private:

Str\* str;

G\_El\* next = nullptr;

G\_El\* prev = nullptr;

public:

G\_El\* getNext();

void setNext(G\_El\* new\_next);

G\_El\* getPrev();

void setPrev(G\_El\* new\_prev);

Str\* getStr();

void setStr(Str\* new\_str);

bool equals(G\_El\* el);

};

Str.cpp

#include "Str.h"

int Str::getLen() {

return len;

};

void Str::setLen(unsigned int new\_len) {

if (new\_len > MAX\_STR\_LEN || new\_len < 0) {

throw "Invalid len";

}

len = new\_len;

};

char Str::getLetter(int i) {

return letters[i];

};

void Str::setLetter(int i, char new\_letter) {

if (i >= MAX\_STR\_LEN || i < 0) {

throw "Invalid index";

}

letters[i] = new\_letter;

};

bool Str::equals(Str\* str) {

if (this == nullptr && str == nullptr) return true;

if (this == nullptr ^ str == nullptr) return false;

if (getLen() != str->getLen()) return false;

for (int i = 0; i < getLen(); i++) {

if (getLetter(i) != str->getLetter(i)) return false;

}

return true;

}

Str.h

#pragma once

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

using namespace std;

const string PROTOCOL\_FILENAME = "protocol.txt";

const int MAX\_STR\_LEN = 20;

class Str {

private:

char letters[MAX\_STR\_LEN];

int len = MAX\_STR\_LEN;

public:

int getLen();

void setLen(unsigned int new\_len);

char getLetter(int i);

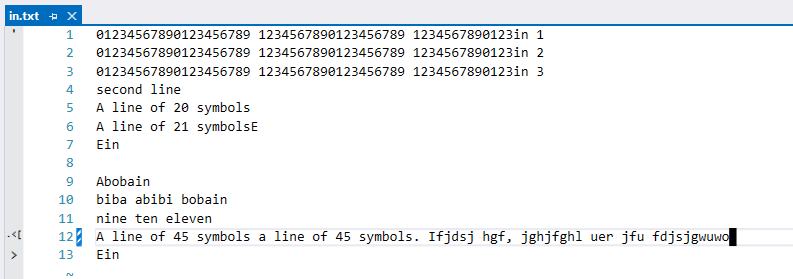
void setLetter(int i, char new\_letter);

bool equals(Str\* str);

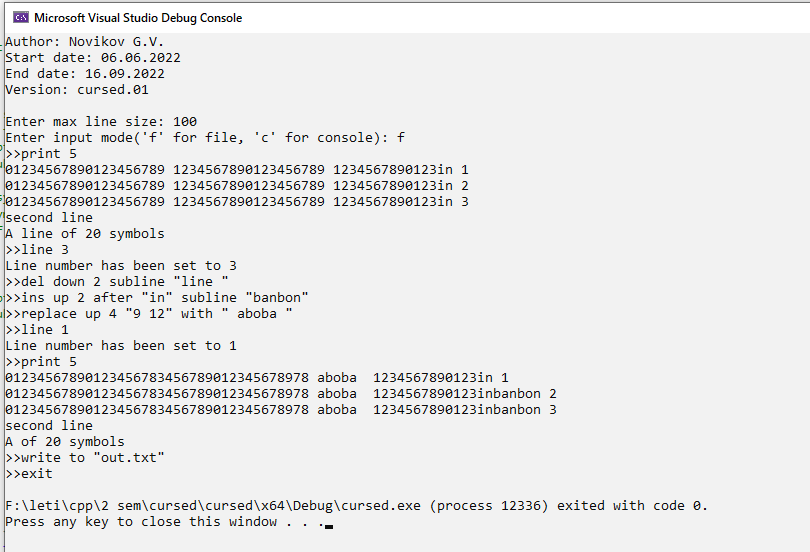
};

# Тестирование программы

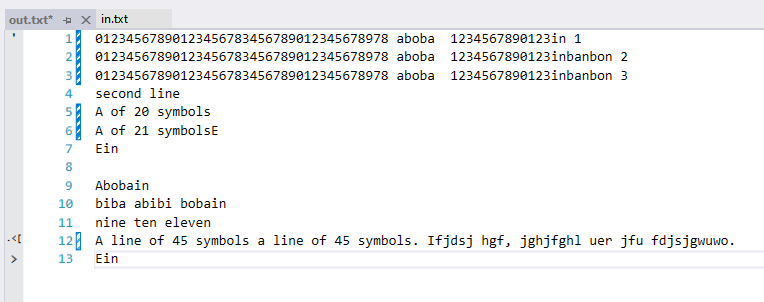
In.txt



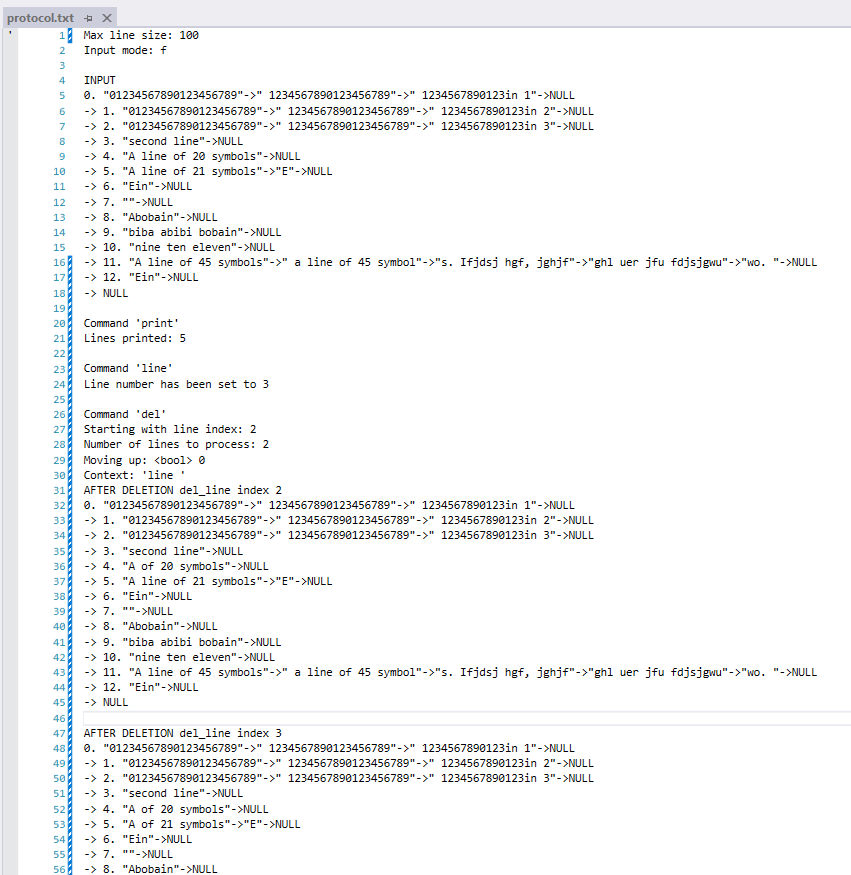
console

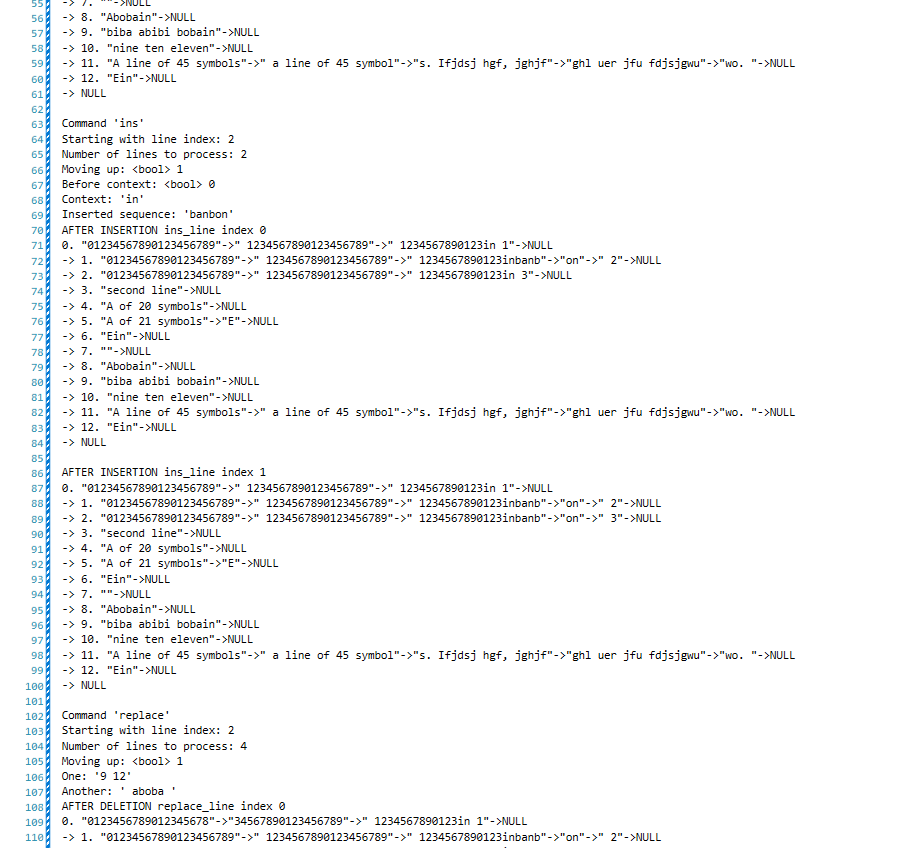


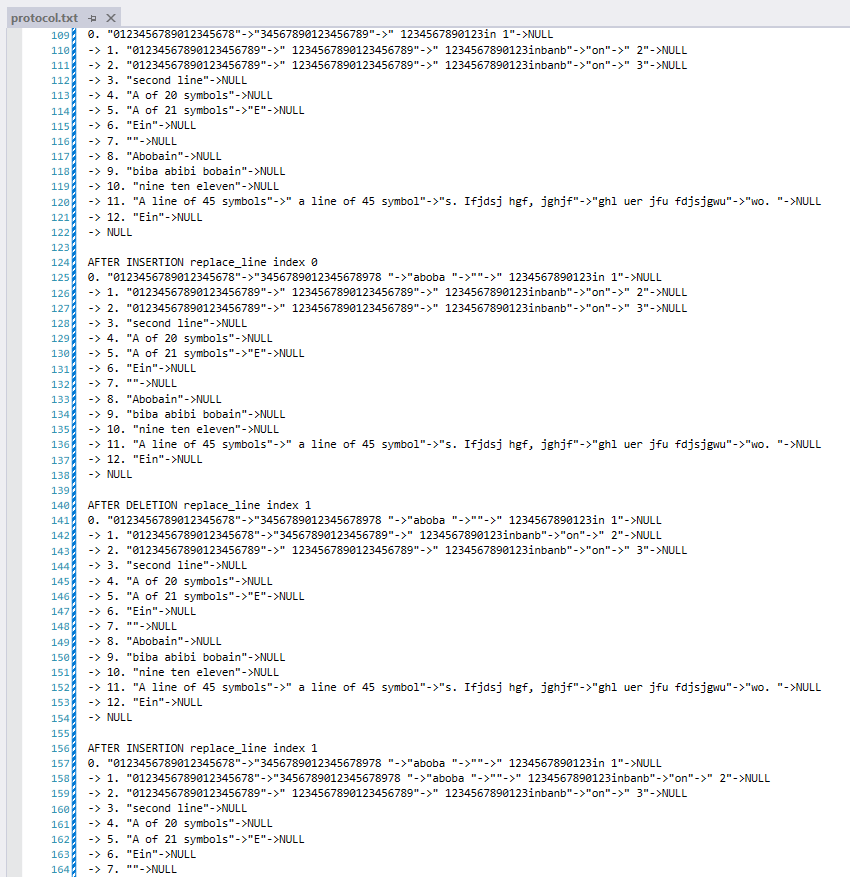
out.txt

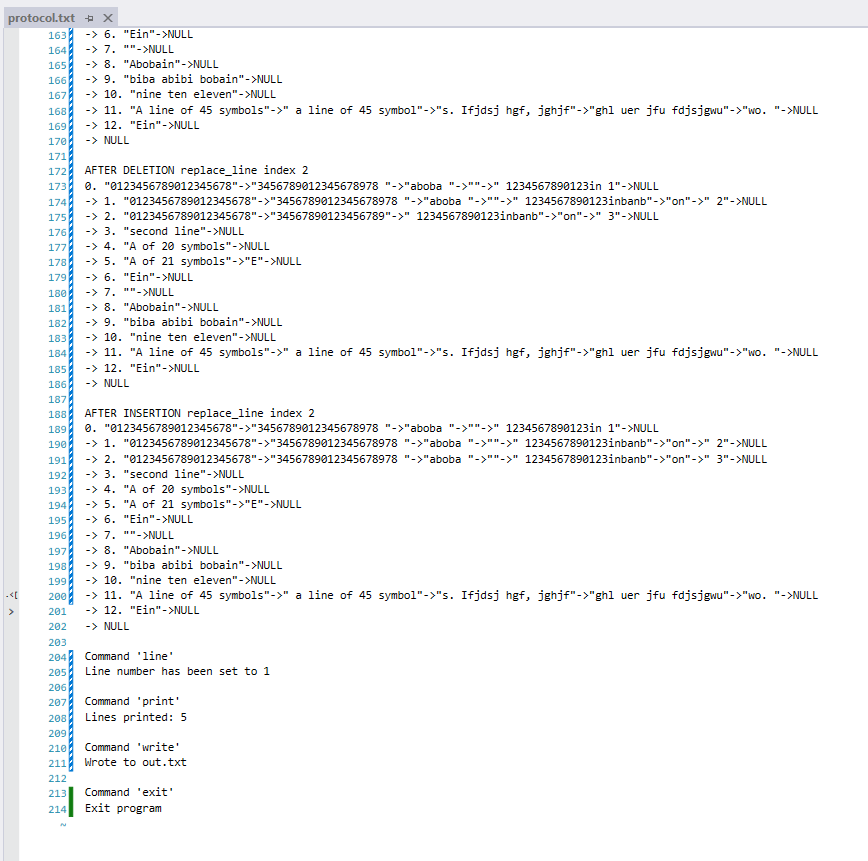


protocol.txt









# Вывод

Цель работы выполнена, программа написана с достаточно удобным интерфейсом. В ходе написания программы были закреплены практические навыки программирования, обобщены знания синтаксиса языка, и продемонстрировано усвоение материала за второй семестр. Поставленная цель полностью соответствует полученному результату.