МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Программирование»

Тема: Обработка текстовых последовательностей

Студент гр. 0382 _____ Шангичев В. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2020

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Шангичев В. А.

Группа 0382

Тема работы: Обработка текстовых последовательностей

Исходные данные:
Программе на вход подается текст (текст представляет собой предложения, разделенные точкой.) Предложения – набор слов, разделенные пробелом или запятой, слова – набор латинских или кириллических букв, цифр и других символов кроме точки, пробела или запятой). Длина текста и кажлого предложения заранее неизвестна.

Для хранения предложения и для хранения текста требуется реализовать структуры Sentence и Text.

Программа должна сохранить (считать) текст в виде динамического массива предложений и оперировать далее только с ним. Функции обработки также должны принимать на вход либо текст (Text), либо предложение (Sentence).

Програма должна найти и удалить все повторно встречающиеся предложения (сравнивать их следует посимвольно, но без учета регистра).

Далее, программа должна запрашивать у пользователя одно из следующих доступных действий (программа должна печатать для этого подсказку. Также следует предусмотреть возможность выхода из программы):

1) Распечатать каждое слово, которое встречается минимум три раза в тексте, а

также количество вхождений каждого такого слова в текст.

2) Заменить каждый символ цифры на число вхождений данной цифры во всем тексте.
3) Отсортировать слова в предложениях по уменьшению количества латинских букв в словах.
4) Поменять порядок слов на обратный в предложениях, которые состоят только из кириллических букв.

Все сортировки и операции со строками должны осуществляться с импользованием функций стандартной библиотеки. Использование собственных функций, при наличии аналога среди функций стандартной библиртеки, запрещается.

Каждую подзадачу следует вынести в отдельную функцию, функции сгруппировать в несколько файлов (например, функции обработки текста в один, функции ввода/вывода в другой). Также, должен быть написан Makefile.

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания: 02.11.2020

Дата сдачи реферата: 16.12.2020

Дата защиты реферата: 22.12.2020

Студент _____ Шангичев В. А.

Преподаватель _____ Жангиров Т. Р.

АННОТАЦИЯ

Для выполнения данной курсовой работы использовался расширенный тип wchar_t для хранения символов, не входящих в таблицу ASCII. Для хранения текста использовалась динамическая память. Также, помимо указанных в задании структур Sentence и Text, для хранения слов была реализована структура Word. Программа получает на вход текст и выполняет с ним действия в соответствии с запросом пользователя. Если запрос был введен некорректно, программа выводит сообщение о некорректности данный. Для тестирования программы на наличие утечек памяти использовалась утилита valgrind.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы состоит в создании работоспособной консольной программы на языке Си, для обработки текста.

Для выпонения цели необходимо решение следующих задач:

- Возможность работы с символами, не входящими в таблицу ASCII.
- Реализация структур для работы с текстом.
- Возможность работы с динамической памятью.

Для выполнения первой задачи используются заголовочные файлы <wctype.h> и <wchar.h>. Для работы с текстом применяется тип wchar_t.

Для выполнения второй задачи были созданы структуры Text, Sentence и Word.

Третья задача реализовывается с помощью заголовочного файла <stdlib.h>. Для выделения памяти используются функции malloc и calloc, для изменения размера выделенной памяти используется функция realloc, для освобождения памяти применяется функция free.

1. Цель и задачи

Цель: разработать консольную программу на языке Си для работы с текстом.

Задачи:

- Считать текст в динамический массив строк и разбить на предложения
- Разбить предложения на слова
- Реализовать ряд функций для выполнения каждой опции программы
- Протестировать работу программы

2. Ход выполнения

2.1. Формат вводимого текста

Текст представляет собой последовательность предложений, разделенных точкой. Предложения состоят из слов, разделенных пробелом или запятой. В тексте может где угодно встречаться символ переноса строки. Ввод данных оканчивается двумя идущими подряд символами переноса строки.

2.2. Структуры для хранения текста

Структуры для хранения текста объявлены в заголовочном файле string_structures.h.

Структура Text имеет следующие поля:

- text_len – количество предложений в введенном тексте
- sentences – массив структур предложений

Структура Sentence включает в себя нижеследующие поля:

- words_number – количество слов в предложении
- separators – разделители между словами
- chars – строка предложения, записанная в исходном виде; после разбиения предложения на слова данное поле очищается.
- words – массив структур слов

Структура Word включает в себя следующие поля:

- symbols – строка слова
- number_of_latin_letters – количество латинских символов в слове; данное поле используется при необходимости.

2.3 Разбиение текста на предложения и слова.

Для считывания текста и разбиения его на предложения используется функция get_text, объявленная в заголовочном файле <functions_for_text.h>. Текст будет считываться в динамический массив строк. Для текущего максимального количества доступной динамической памяти для сохранения строк и символов в них используются переменные current_max_text_len и

current_max_sentence_len соответственно. Индексы динамического массива, по которым определяется ячейка, в которую надо считать следующий символ, хранятся в переменных current_symbol_index и current_sentence_index. После выделения динамической памяти идет цикл while, условием выхода из которого является равенство текущего символа и символа перевода строки и значение переменной is_enter 1. Значение переменной is_enter устанавливается в нуль, если текущий символ не является символом перевода строки и устанавливается в единицу, если текущий символ является символом перевода строки. Таким образом, после двух подряд идущих символов перевода строки будет осуществляться выход из цикла. Текущий символ записывается в соответствующую ячейку двумерного массива text, после чего переменная current_symbol_index увеличивается на один. Далее выполняется проверка на равенство текущего индекса и текущей максимальной длины строки. Если условие истинно, выполняется перевыделение памяти для каждой строки, в которую не осуществлялась запись, включая текущую. Затем проверяется, не является ли текущий символ точкой. Если условие истинно, в массив строк записывается нулевой символ. Затем необходимо проверить, не встречалась ли данная строка ранее. Данное действие выполняется с помощью функции wcscasecmp. Сравнение выполняется со всеми ранее записанными строками. Если функция возвращает нуль, значит строки равны – все символы обнуляются (данное действие необязательно), и значение индекса текущей строки остается прежним. Если же данное предложение встречается в первый раз, то значение индекса текущего предложения увеличивается на один. Последняя проверка в цикле гарантирует, что длина массива строк достаточна для записи нового предложения. При выходе из цикла освобождаются все указатели массива строк, которые не понадобились для записи предложений. После чего выделяется динамическая память для хранения структур Sentence. Далее в цикле полю chars каждой структуры присваивается значение соответствующего указателя в массиве строк. После записи, вся память для хранения указателей на строки освобождается (указатели на строки теперь содержатся в структурах). Сответствующим полям структуры текст присваивается количество предложений и указатель на структуры.

Для разбиения текста на слова реализуется функция get_words. Сначала выделяются три блока динамической памяти. Первый для хранения массивов разделителей для каждого предложения, второй для массивов слов для каждого предложения и третий для хранения символов, содержащихся в каждом слове. После выделения памяти идет цикл for, перебирающий индексы всех предложений в тексте. В этот цикл вложен бесконечный цикл с переменной j, увеличивающейся на один на каждой итерации. В переменную current_symbol записывается символ соответствующего предложения. Этот символ присваивается соответствующей ячейке массива symbols_buffer, после чего значение индекса текущего символа этого массива увеличивается на один. Затем выполняется проверка на наличие свободного места в массиве symbols_buffer для записи следующего символа. Если места нет, для каждой

строки, начиная с текущей, выделяется дополнительная память. Далее идет проверка на равенство текущего символа с пробелом или запятой. Если условие истинно, то в массив разделителей записывается текущий символ, а в массив symbols_buffer вместо пробела или запятой записывается нулевой символ. Полю symbols соответствующей структуре в массиве структур присваивается текущий указатель на строку в массиве symbols_buffer. Индекс текущего символа в массиве symbols_buffer обнуляется. Далее выполняется проверка на наличие свободного места на запись следующего разделителя в массиве separators. Если условие истинно, память перевыделяется. После этого выполняется проверка на равенство текущего символа с точкой. Если условие истинно, то на место точки в массив symbols_buffer записывается нулевой символ. Полю соответствующей структуры в массиве st_words_buffer присваивается соответствующая строка из массива symbols_buffer. Затем выполняется проверка, имеется ли в массиве структур указатель для записи слов для следующего предложения. Если нет, то память перевыделяется. Значение переменной in_sentence устанавливается в нуль. Далее полю words соответствующего предложения присваивается соответствующий указатель из массива st_words_buffer, полю words_number количество слов в данном предложении, и полю separators присваивается соответствующая строка из массива separators. Далее идут три последних условия: на достаточность строк для записи разделителей следующего предложения в массиве separators, на достаточность длины массива структур для записи текущего слова и на достаточность количества строк в массиве symbols_buffer для записи следующего символа. Если где-либо условие выполняется, память перевыделяется. В конце выполняется проверка значения переменной in_sentence. Если значение равно нулю, то осуществляется выход из внутреннего цикла, и все вышеперечисленные действия выполняются для следующего предложения. После окончания работы цикла освобождается память, непонадобившаяся для записи слов, а также память для всех указателей на память, значения которых теперь записаны в структурах Word и Sentence.

2.4 Реализация опции 1.

Для реализации первой опции программы была создана вспомогательная структура Substrings. Поле words хранит все уникальные слова в программе, поле num_occur хранит число, соответствующее числу раз встречаемости слова в тексте и поле num_words хранит количество уникальных слов в тексте. Функционал опции 1 реализовывает функция get_substrings. В теле функции выделяется память для хранения уникальных слов и количеству раз встречаемости каждого такого слова. С помощью двух циклов for перебираются все слова в тексте. В переменную current_word записывается строка, хранящая текстовое представление текущего слова. Далее осуществляется сравнение со всеми ранее записанными уникальными словами. Если данное слово уже было записано, то соответствующая переменная в массиве occuriences увеличивается на один, и работа цикла сравнения слов завершается. Далее выполняется проверка на равенство индекса k и текущему количеству уникальных слов.

Если условие истинно, значит, данное слово еще не было записано, и является уникальным. При необходимости для хранения количества встречаемости данного слова выделяется память. Далее выполняется проверка на наличие строки для записи данного слова. При необходимости создаются новые строки. Затем необходимо проверить, достаточна ли текущая длина строки для записи текущего слова, включая нулевой символ. Если при удваивании текущей длины строки места для записи текущего слова все еще не будет хватать, то размер памяти увеличивается до необходимого. Если нет, то размер удваивается. Символы строки копируются в массив уникальных слов, соответствующая ячейка в массиве occuriences увеличивается на один, значение переменной num_words также увеличивается на один. После перебора всех слов освобождается непонадобившаяся память, и соответствующим полям структуры substrings присваиваются соответствующие значения. По завершении работы функции get_substrings выводятся все слова, для которых соответствующее значение в массиве occuriences больше 3. Если таких слов нет, то выводится сообщение “По Вашему запросу ничего не найдено”. Память, необходимая для хранения полей объекта структуры Substrings освобождается.

2.5 Реализация опции 2.

За реализацию опции 2 отвечает функция replace_digits. В теле данной функции сначала выделяется память под массив строк words_buffer. В данный массив запишутся новые строковые представления всех слов. Также объявляется статический массив array для хранения встречаемости каждой цифры в тексте. С помощью трех циклов перебираются все символы текста. С помощью функции iswdigit проверяется, является ли текущий символ цифрой. Если да, то переменная, хранящая количество цифр, увеличивается на один. Из значения текущего символа вычитается код символа нуль, полученный результат позволяет получить целочисленное значение строкового представления цифры. Соответствующая ячейка массива array увеличивается на один. После получения количества встречаемости для каждой цифры, необходимо создать строковое представление каждого количества встречаемости. Для этого с помощью цикла высчитывается количество разрядов в значении переменной num_digits (это пограничный случай, если все цифры в тексте одинаковые). После этого создается двумерный массив strings_for_replace. Каждой из десяти строк массива присваивается результат работы функции swprintf для каждого числа в массиве array. Функция swprintf по сути делает то же, что и функция wprintf, только она не выводит символы на экран, а сохраняет их в указанной строке. Далее снова перебираются все символы в тексте. Текущий символ сохраняется в массиве words_buffer. Если текущий символ является цифрой, то сохраненный в массиве words_buffer символ заменяется нулевым символом, после чего измеряется длина строки, на которую будет заменена текущая цифра. До тех пор, пока длина буфера недостаточна для записи текущего слова и строки из strings_for_replace, размер буфера увеличивается. После чего строковое представление текущего слова и строки, на которую заменяется цифра конкатенируются. Значение переменной

b_index увеличивается так, чтобы следующий символ был записан на место нулевого символа. Далее значения индекса текущего символа увеличивается на один и выполняется проверка на достаточность места для сохранения следующего символа. Если места недостаточно, то выделяется дополнительная память. После завершения работы цикла while освобождается старое строковое представление текущего слова, в конец соответствующего слова в массиве words_buffer добавляется нулевой символ, полю symbols структуры word присваивается новое строковое представление. Далее выполняется проверка на наличие строки для сохранения строкового представления следующего слова. При необходимости выделяется дополнительная память. После завершения работы цикла освобождается память для хранения всех массивов. После работы функции весь текст выводится на экран с помощью функции print_text, которая описывается в разделе 2.8.

2.6 Реализация опции 3.

Для начала для каждого слова в тексте вычисляется количество латинских символов с помощью функции get_number_of_latin_letters. Данная функция перебирает все символы слова и проверяет каждый из них на принадлежность к диапозону заглавных и строчных латинских букв. Возвращает количество символов, прошедших проверку. После этого вызывается функция sort_words, которая сортирует слова внутри каждого предложения с помощью функции qsort. Функция компаратор – compare сравнивает значения поля number_of_latin_letters двух переданных слов. После сортировки на экран выводится весь текст с помощью функции print_text.

2.7 Реализация опции 4.

Для реализации опции 4 используется функция set_k. В ней для каждого предложения вызывается функция check_kirl. В этой функции проверяются все символы переданного предложения. Если хотя бы один из них не является кириллическим, то программа возвращает значение 0. В противном случае – значение 1. Если функция вернула 1, то для текущего предложения вызывается функция reverse. В этой функции с помощью перемены местами соответствующих элементов массива массив words предложения s переворачивается. Измененный текст выводится на экран с помощью функции print_text.

2.8 Очистка динамической памяти перед завершением программы, реализация вывода меню и текста.

Перед завершением работы программы для очистки памяти вызывается функция free_memory. В ней освобождается динамическая память, выделенная для хранения символов слов, разделителей предложений, структур слов и структур предложений. Вывод меню осуществляется с помощью функции print_menu. Текст выводится с помощью функции print_text. В данной функции сначала выводится строковое представления текущего слова, затем печатается разделитель. Каждое предложение выводится с новой строки.

2.9 Разбиение функций на файлы

Для удобства код разбит на файлы следующим образом:

- заголовочный файл consts.h содержит все макроопределения, задающие начальный размер буфера;
- файл functions_for_sentence.c содержит описание всех функций для работы с предложениями;
- заголовочный файл functions_for_sentence.h содержит определения всех функций для работы с предложениями;
- файл functions_for_text.c содержит описание всех функций для работы с текстом;
- заголовочный файл functions_for_text.h содержит определение всех функций для работы с текстом;
- файл main.c является главным файлом программы и содержит вызов всех функций, необходимых для работы программы;
- файл string_functions.c содержит вспомогательные функции для сортировки/выделения подстрок текста;
- заголовочный файл содержит все определения функций из файла string_functions.h;
- заголовочный файл string_structures.h содержит определения структур Text, Sentence и Word;
- заголовочный файл substring_struct.h содержит описание структуры Substrings;

Также был написан Makefile, при помощи которого для каждого файла создается объектный файл, после чего создается исполняемый файл программы с именем program.

ЗАКЛЮЧЕНИЕ

Была разработана программа для обработки текста. Для решения задачи использовались навыки работы с динамической памятью и расширенной таблицей символов. Работа программы была протестирована с помощью утилиты valgrind.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Сайт stackoverflow.com (дата обращения 12.12.2020)
2) Керниган Б. и Ритчи Д. Язык программирования Си. М.: Вильямс, 1978

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ ПРОГРАММЫ

№	ввод	Вывод (вывод меню опущен)	Вывод утилиты valgrind	Комментарий
1	Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» имени В.И.Ульянова (Ленина) — один из ведущих старейших российских технических высших учебных заведений,основанный в 1886 году,был первым в Европе высшим учебным заведением,специализированным в области электротехники. 1	в 3	in use at exit: 0 bytes in 0 blocks. 0 errors from 0 contexts.	Программа работает корректно.
2	The first major advance in abstraction was the use of numerals to represent numbers.This allowed systems to be developed for recording large numbers.The ancient Egyptians developed a powerful system of numerals with distinct hieroglyphs for 1,10,and all powers of 10 up to over 1 million.A stone carving from Karnak,dating back from around 1500 BCE and now at the Louvre in Paris,depicts 276 as 2 hundreds,7 tens,and 6 ones.and similarly for the number 4,622. The Babylonians had a place-value system based essentially on the numerals for 1 and 10,using base sixty,so that the symbol for sixty was the same as the symbol for one—its value being determined from context. 2	The first major advance in abstraction was the use of numerals to represent numbers.This allowed systems to be developed for recording large numbers.The ancient Egyptians developed a powerful system of numerals with distinct hieroglyphs for 7,75,and all powers of 75 up to over 7 million.A stone carving from Karnak,dating back from around 7155 BCE and now at the Louvre in Paris,depicts 423 as 4 hundreds,2 tens,and 3 ones.and similarly for the number 1,344. The Babylonians had a place-value system based essentially on the numerals for 7 and 75,using base	in use at exit: 0 bytes in 0 blocks. 0 errors from 0 contexts.	Программа работает корректно.

		sixty,so that the symbol for sixty was the same as the symbol for one—its value being determined from context.		
3	Одинаковые предложения.Одинаковые предложения.Здесь mnogo latinskih simvolov.Здесь латинских символов malo. 3	Строка номер 1 Одинаковые предложения Строка номер 2 latinskih simvolov mnogo Здесь Строка номер 3 malo Здесь латинских символов	in use at exit: 0 bytes in 0 blocks. 0 errors from 0 contexts.	Программа работает корректно.
4	Это предложение полностью состоит из кириллических символов.This sentence not. 4	Строка номер 1 символов кириллических из состоит полностью предложение Это Строка номер 2 This sentence not	in use at exit: 0 bytes in 0 blocks. 0 errors from 0 contexts.	Программа работает корректно.

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

Весь код находится в файлах в папке /src/

Файл consts.h

```
#ifndef CN
#define CN

#define TEXTSIZE 32
#define BUFSIZE 128
#define MAXSTWORDS 32
#define MAXSTLEN 128
#define MAXSYWORDS 32
#define MAXSYLEN 128
#define ARRAYBUF 128

#endif
```

Файл functions_for_sentence.c

```
#include <stdio.h>
#include <wchar.h>
#include "functions_for_sentence.h"

int check_kirl(struct Sentence* s){
    // проверяет, состоит ли предложение только из кириллических букв
    int i, j;
    wchar_t current_symbol;

    for (i = 0; i < s->words_number; i++){
        j = 0;
        while ((current_symbol = s->words[i].symbols[j++]) != '\0'){
            if ((current_symbol >= L'a' && current_symbol <= L'я') ||
(current_symbol >= L'A' && current_symbol <= L'Я')){
                continue;
            } else {
                return 0;
            }
        }
```

```
    }
    return 1;
}

void reverse(struct Sentence* s){
    struct Word word;
    int i;
    int last_index = (s->words_number / 2 - 1);
    for (i = 0; i <= last_index; i++){
        word = s->words[i];
        s->words[i] = s->words[s->words_number - i - 1];
        s->words[s->words_number - i - 1] = word;
    }
}
```

Файл functions_for_sentence.h

```
#ifndef FFSE
#define FFSE
#include "string_structures.h"

int check_kirl(struct Sentence* s);
void reverse(struct Sentence* s);
#endif
```

Файл functions_for_text.c

```
#include <stdio.h>
#include <wchar.h>
#include "consts.h"
#include <stdlib.h>
#include "functions_for_text.h"
#include <wctype.h>
#include "string_functions.h"

void free_memory(struct Text* text){
    int i, j;
    for (i = 0; i < text->text_len; i++){
        for (j = 0; j < text->sentences[i].words_number; j++){
            free(text->sentences[i].words[j].symbols);
```

```
        }
        free(text->sentences[i].words);
        free(text->sentences[i].separators);
    }
    free(text->sentences);
}

void get_words(struct Text* text) {
    int in_sentence = 1;
    wchar_t** p_for_sym_buf;
    wchar_t* p_for_sym;
    struct Word** p_for_st_words_b;
    struct Word* p_for_st;
    int max_stwords = MAXSTWORDS;
    int max_stlen = MAXSTLEN;
    int max_sy_words = MAXSYWORDS;
    int max_sy_len = MAXSYLEN;
    int max_sep_arlen = ARRAYBUF;
    int max_seps_num = ARRAYBUF;
    int struct_index = 0;
    int sentence_index = 0;
    int word_index = 0;
    int symbol_index = 0;
    int sep_index = 0;
    int i, j, k;
    wchar_t current_symbol;

    wchar_t** separators = malloc(sizeof(wchar_t*) * max_sep_arlen);
    for (i = 0; i < max_sep_arlen; i++){
        separators[i] = malloc(sizeof(wchar_t) * max_seps_num);
    }

    struct Word** st_words_buffer = malloc(sizeof(struct Word*) *
max_stwords);
    for (i = 0; i < max_stwords; i++){
        st_words_buffer[i] = malloc(sizeof(struct Word) * max_stlen);
    }
```

```
    wchar_t** symbols_buffer = malloc(sizeof(wchar_t*) * max_sy_words);
    for (i = 0; i < max_sy_words; i++){
        symbols_buffer[i] = malloc(sizeof(wchar_t) * max_sy_len);
    }


    for (i = 0; i < text->text_len; i++){
        in_sentence = 1;
        for (j = 0;;j++){
            current_symbol = text->sentences[i].chars[j];
            symbols_buffer[word_index][symbol_index] = current_symbol;
            symbol_index += 1;

            if (symbol_index == max_sy_len){
                max_sy_len *= 2;
                for (k = word_index; k < max_sy_words; k++){
                    p_for_sym = realloc(symbols_buffer[k],
sizeof(wchar_t) * max_sy_len);
                    if (p_for_sym){
                        symbols_buffer[k] = p_for_sym;
                    }
                }
            }
            if (current_symbol == L' ' || current_symbol == L','){
                // end of word
                separators[sentence_index][sep_index++] = current_symbol;
                symbols_buffer[word_index][symbol_index-1] = '\0';
                st_words_buffer[sentence_index][struct_index].symbols =
symbols_buffer[word_index];
                struct_index += 1;
                word_index += 1;
                symbol_index = 0;
            }

            if (sep_index == max_seps_num){
                max_seps_num *= 2;
                for (k = sentence_index; k < max_sep_arlen; k++){
```

```
                    p_for_sym = realloc(separators[k], sizeof(wchar_t) *
max_seps_num);
                    if (p_for_sym){
                        separators[k] = p_for_sym;
                    }
                }
            }

            if (current_symbol == L'.'){
                symbols_buffer[word_index][symbol_index-1] = '\0';
                st_words_buffer[sentence_index][struct_index].symbols =
symbols_buffer[word_index];
                symbol_index = 0;
                word_index += 1;
                sentence_index += 1;
                if (sentence_index == max_stwords){
                    max_stwords *= 2;
                    p_for_st_words_b = realloc(st_words_buffer,
sizeof(struct Word*) * max_stwords);
                    if (p_for_st_words_b){
                        st_words_buffer = p_for_st_words_b;
                    }
                    for (k = sentence_index; k < max_stwords; k++){
                        st_words_buffer[k] = malloc(sizeof(struct Word) *
max_stlen);
                    }

                }
                text->sentences[i].words =
st_words_buffer[sentence_index-1];
                text->sentences[i].words_number = struct_index + 1;
                text->sentences[i].separators =
separators[sentence_index-1];
                sep_index = 0;
                struct_index = 0;
                in_sentence = 0;
            }
```

```
            if (sentence_index == max_sep_arlen){
                max_sep_arlen *= 2;
                p_for_sym_buf = realloc(separators, sizeof(wchar_t*) *
max_sep_arlen);
                if (p_for_sym_buf){
                    separators = p_for_sym_buf;
                }
                for (k = sentence_index; k < max_sep_arlen; k++){
                    p_for_sym = realloc(separators[k], sizeof(wchar_t) *
max_seps_num);
                    if (p_for_sym){
                        separators[k] = p_for_sym;
                    }
                }
            }

            if (struct_index == max_stlen){
                max_stlen *= 2;
                for (k = sentence_index; k < max_stwords; k++){
                    p_for_st = realloc(st_words_buffer[k], sizeof(struct
Word) * max_stlen);
                    if (p_for_st){
                        st_words_buffer[k] = p_for_st;
                    }
                }
            }
            if (word_index == max_sy_words){
                max_sy_words *= 2;
                p_for_sym_buf = realloc(symbols_buffer, sizeof(wchar_t*)
* max_sy_words);
                if (p_for_sym_buf){
                    symbols_buffer = p_for_sym_buf;
                }
                for (k = word_index; k < max_sy_words; k++){
                    symbols_buffer[k] = malloc(sizeof(wchar_t) *
max_sy_len);
                }
            }
```

```
            if (!in_sentence){
                break;
            }
        }
    }

    for (k = sentence_index; k < max_sep_arlen; k++){
        free(separators[k]);
    }

    for (k = word_index; k < max_sy_words; k++){
       free(symbols_buffer[k]);
    }
    for (k = sentence_index; k < max_stwords; k++){
        free(st_words_buffer[k]);
    }
    for (k = 0; k < text->text_len; k++){
        free(text->sentences[k].chars);
    }
    free(separators);
    free(symbols_buffer);
    free(st_words_buffer);

}

void get_text(struct Text* processed_text){
    int is_enter = 0;
    int is_equal = 1;
    wchar_t** p_for_text;
    wchar_t* p_for_sentence;
    int i, j;
    wchar_t current_symbol;
    int current_symbol_index = 0;
    int current_sentence_index = 0;
    int current_max_text_len = TEXTSIZE;
    int current_max_sentence_len = BUFSIZE;
```

```
    wchar_t** text = (wchar_t**)malloc(sizeof(wchar_t*) *
current_max_text_len);

    for (i = 0; i < current_max_text_len; i++){
        text[i] = (wchar_t*)malloc(sizeof(wchar_t) *
current_max_sentence_len);
    }

    // reading text
    while ((current_symbol = getwchar()) != L'\n' || is_enter == 0){
        if (current_symbol == L'\n'){
            is_enter = 1;
            continue;
        }
        is_enter = 0;
        text[current_sentence_index][current_symbol_index] =
current_symbol;
        current_symbol_index += 1;

        if (current_symbol_index == current_max_sentence_len){
            // add memory
            current_max_sentence_len *= 2;
            for (i = current_sentence_index; i < current_max_text_len;
i++){
                p_for_sentence = (wchar_t*)realloc(text[i],
sizeof(wchar_t) * current_max_sentence_len);
                if (p_for_sentence){
                    text[i] = p_for_sentence;
                }
            }
        }
        if (current_symbol == '.'){
            // end of sentence
            text[current_sentence_index][current_symbol_index] = '\0';
            // check string
            for (i = 0; i < current_sentence_index; i++){
                is_equal = wcscasecmp(text[i],
text[current_sentence_index]);
```

```
                if (!is_equal){
                    // не добавлять строку
                    for (j = 0; j <= current_symbol_index; j++){
                        text[current_sentence_index][j] = 0;
                    }
                    break;
                }
            }
            current_symbol_index = 0;
            if (is_equal){
                current_sentence_index += 1;
            }
        }

        if (current_sentence_index == current_max_text_len){
            // add memory
            current_max_text_len *= 2;
            p_for_text = (wchar_t**)realloc(text, sizeof(wchar_t*) *
current_max_text_len);
            if (p_for_text){
                text = p_for_text;
            }
            for (i = current_sentence_index; i < current_max_text_len;
i++){
                text[i] = (wchar_t*)malloc(sizeof(wchar_t) *
current_max_sentence_len);
            }
        }
    }

    // free memory
    for (i = current_sentence_index; i < current_max_text_len; i++){
        free(text[i]);
    }

    struct Sentence* sentences = malloc(sizeof(struct Sentence) *
current_sentence_index);
```

```
    // init structures
    for (i = 0; i < current_sentence_index; i++){
        sentences[i].chars = text[i];
    }
    free(text);
    processed_text->sentences = sentences;
    processed_text->text_len = current_sentence_index;
}


void print_text(struct Text* text){
    int i, j, k;
    for (i = 0; i < text->text_len; i++){
        k = 0;

        for (j = 0; j < text->sentences[i].words_number; j++){
            wprintf(L"%ls", text->sentences[i].words[j]);
            if (k != text->sentences[i].words_number - 1){
                wprintf(L"%lc", text->sentences[i].separators[k++]);
            }
        }
        wprintf(L".\n");
    }
}

void replace_digits(struct Text* text){
    int is_digit;
    int max_digit_for_replace_len = 0;
    int index;
    int array[10];
    wchar_t current_char;
    int i, j, k, w;
    for (i = 0; i < 10; i++){
        array[i] = 0;
    }
    int num_digits = 0;
    int max_wb_len = ARRAYBUF;
    int max_b_len = ARRAYBUF;
```

```
    wchar_t** wb_p;
    wchar_t* b_p;
    int wb_index = 0;
    int b_index = 0;
    int str_len;
    wchar_t current_symbol;
    wchar_t** words_buffer = malloc(sizeof(wchar_t*) * max_wb_len);
    for (i = 0; i < max_wb_len; i++){
        words_buffer[i] = malloc(sizeof(wchar_t) * max_b_len);
    }

    for (i = 0; i < text->text_len; i++){
        for (j = 0; j < text->sentences[i].words_number; j++){
            k = 0;
            while (text->sentences[i].words[j].symbols[k] != '\0'){
                is_digit = iswdigit(text-
>sentences[i].words[j].symbols[k]);
                if (is_digit){
                    num_digits++;
                    index = text->sentences[i].words[j].symbols[k] -
L'0';
                    array[index]++;
                }
                k++;
            }
        }
    }
    while (num_digits > 0){
        max_digit_for_replace_len++;
        num_digits /= 10;
    }

    wchar_t** strings_for_replace = malloc(sizeof(wchar_t*) * 10);
    for (i = 0; i < 10; i++){
        strings_for_replace[i] = malloc(sizeof(wchar_t) *
(max_digit_for_replace_len + 1));
    }
    for (i = 0; i < 10; i++){
```

```
        swprintf(strings_for_replace[i], max_digit_for_replace_len + 1,
L"%d", array[i]);
    }

    for (i = 0; i < text->text_len; i++){
        for (j = 0; j < text->sentences[i].words_number; j++){
            k = 0;
            while (text->sentences[i].words[j].symbols[k] != '\0'){
                current_symbol = text->sentences[i].words[j].symbols[k];
                words_buffer[wb_index][b_index] = current_symbol;
                is_digit = iswdigit(current_symbol);
                k++;
                if (is_digit){
                    words_buffer[wb_index][b_index] = '\0';
                    str_len = wcslen(strings_for_replace[current_symbol -
L'0']);
                    while (max_b_len - b_index - 1 < str_len){
                        max_b_len *= 2;
                        for (w = wb_index; w < max_wb_len; w++){
                            b_p = realloc(words_buffer[w],
sizeof(wchar_t) * max_b_len);
                            if (b_p){
                                words_buffer[w] = b_p;
                            }
                        }
                    }
                    wcscat(words_buffer[wb_index],
strings_for_replace[current_symbol - L'0']);
                    b_index += str_len - 1;
                }

                b_index++;
                if (b_index == max_b_len){
                    max_b_len *= 2;
                    for (w = wb_index; w < max_wb_len; w++){
                        b_p = realloc(words_buffer[w], sizeof(wchar_t) *
max_b_len);
                        if (b_p){
```

```
                            words_buffer[w] = b_p;
                        }
                    }
                }

            }
            free(text->sentences[i].words[j].symbols);
            words_buffer[wb_index][b_index] = '\0';
            text->sentences[i].words[j].symbols =
words_buffer[wb_index++];
            b_index = 0;
            if (wb_index == max_wb_len){
                max_wb_len *= 2;
                wb_p = realloc(words_buffer, sizeof(wchar_t*) *
max_wb_len);
                if (wb_p){
                    words_buffer = wb_p;
                }
                for (w = wb_index; w < max_wb_len; w++){
                    words_buffer[w] = malloc(sizeof(wchar_t) *
max_b_len);
                }
            }


        }
    }

    for (i = wb_index; i < max_wb_len; i++){
        free(words_buffer[i]);
    }
    for (i = 0; i < 10; i++){
        free(strings_for_replace[i]);
    }
    free(strings_for_replace);
    free(words_buffer);
}
```

```
void sort_words(struct Text* text){
    int i;

    for (i = 0; i < text->text_len; i++){
        qsort(text->sentences[i].words, text->sentences[i].words_number,
sizeof(struct Word), compare_words);
    }
}
```

Файл functions_for_text.h

```
#ifndef FFT
#define FFT

#include "string_structures.h"
#include "substring_struct.h"

void free_memory(struct Text* text);
void get_text(struct Text* processed_text);
void get_words(struct Text* text);
void replace_digits(struct Text* text);
void sort_words(struct Text* text);
void print_text(struct Text* text);

#endif
```

Файл main.c

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>
#include "consts.h"
#include "string_structures.h"
#include "substring_struct.h"
#include "functions_for_sentence.h"
#include "functions_for_text.h"
#include "string_functions.h"

void print_menu(void);
```

```
int main() {
    setlocale(LC_ALL, "");
    struct Text processed_text;
    int user_request;
    int i, j;
    struct Substrings answer;
    int is_correct;
    get_text(&processed_text);
    get_words(&processed_text);
    wchar_t current_symbol;
    int is_find;

    // вывод меню

    for (;;){
        // запрос от пользователя
        print_menu();
        user_request = 0;
        wscanf(L"%d", &user_request);
        while ((current_symbol = getwchar()) != '\n');
        if (user_request == 5){
            free_memory(&processed_text);
            return 0;
        }

        switch (user_request){
            case 1:
                is_find = 0;
                get_substrings(&answer, &processed_text);
                for (i = 0; i < answer.num_words; i++){
                    if (answer.num_occur[i] > 2){
                        is_find = 1;
                        wprintf(L"%ls %d\n", answer.words[i],
answer.num_occur[i]);
                    }
                    free(answer.words[i]);
                }
```

```
                free(answer.words);
                free(answer.num_occur);
                if (!is_find){
                    wprintf(L"По Вашему запросу ничего не найдено.");
                }
                break;
            case 2:
                replace_digits(&processed_text);
                print_text(&processed_text);
                break;
            case 3:
                for (i = 0; i < processed_text.text_len; i++){
                    for (j = 0; j <
processed_text.sentences[i].words_number; j++){

processed_text.sentences[i].words[j].number_of_latin_letters =
get_number_of_latin_letters(processed_text.sentences[i].words[j]);
                    }
                }

                sort_words(&processed_text);
                print_text(&processed_text);
                break;
            case 4:
                set_k(&processed_text);
                print_text(&processed_text);
                break;
            default:
                wprintf(L"Неизвестная команда.\n");
                break;
        }
    }

    return 0;
}

void print_menu(void){
    wprintf(L"\n\nВыберите функцию обработки текста:\n");
```

```
    wprintf(L"1 - вывод всех слов, встречающихся в тексте более двух
раз\n");
    wprintf(L"2 - замена всех цифр на число их вхождений\n");
    wprintf(L"3 - сортировка слов в предложениях по числу латинских букв
в них\n");
    wprintf(L"4 - изменить порядок слов на обратный в предложениях,
состоящих только из кириллических букв\n");
    wprintf(L"5 - выход\n");
}
```

Файл Makefile

```
all: main.o string_functions.o functions_for_text.o
functions_for_sentence.o
     gcc main.o string_functions.o functions_for_text.o
functions_for_sentence.o -o program

main.o: main.c
     gcc -c main.c

string_functions.o: string_functions.c
     gcc -c string_functions.c

functions_for_text.o: functions_for_text.c
     gcc -c functions_for_text.c

functions_for_sentence.o: functions_for_sentence.c
     gcc -c functions_for_sentence.c

clean:
     rm -f *.o
```

Файл string_functions.c

```
#include <wchar.h>
#include "string_functions.h"
#include "consts.h"
#include <stdlib.h>
#include <wctype.h>
#include "functions_for_sentence.h"
```

```
#include <stdio.h>

void get_substrings(struct Substrings* substrings, struct Text* text){
    // create two arrays:
    // words and num occuriences
    wchar_t* current_word;
    int next_max_word_len;
    struct Sentence current_sentence;
    int i, j, k, w, difference, word_len;
    int* p_for_occuriences;
    wchar_t** p_for_warray;
    wchar_t* p_for_word;
    int is_equal;
    int num_words = 0;
    int max_array_len = ARRAYBUF;
    int max_words_array_len = MAXSYWORDS;
    int max_word_len = MAXSYLEN;
    int* occuriences = calloc(max_array_len, sizeof(int));
    wchar_t** words = malloc(sizeof(wchar_t*) * max_words_array_len);
    for (i = 0; i < max_words_array_len; i++){
        words[i] = malloc(sizeof(wchar_t) * max_word_len);
    }
    for (i = 0; i < text->text_len; i++){
        current_sentence = text->sentences[i];
        for (j = 0; j < current_sentence.words_number; j++){
            current_word = current_sentence.words[j].symbols;
            for (k = 0; k < num_words; k++){
                is_equal = wcscmp(current_word, words[k]);
                if (!is_equal){
                    occuriences[k]++;
                    break;
                }
            }
            if (k == num_words){
                // create new word
                // check memory
                // for occuriences
                if (k == max_array_len){
```

```
                    // add memory
                    max_array_len *= 2;
                    p_for_occuriences = realloc(occuriences, sizeof(int)
* max_array_len);
                    if (p_for_occuriences){
                        occuriences = p_for_occuriences;
                    }
                    for (w = k; w < max_array_len; w++){
                        occuriences[w] = 0;
                    }
                }
                // for words array
                if (k == max_words_array_len){
                    max_words_array_len *= 2;
                    p_for_warray = realloc(words, sizeof(wchar_t*) *
max_words_array_len);
                    if (p_for_warray){
                        words = p_for_warray;
                    }
                    for (w = num_words; w < max_words_array_len; w++){
                        p_for_word = malloc(sizeof(wchar_t) *
max_word_len);
                        if (p_for_word){
                            words[w] = p_for_word;
                        }
                    }

                }
                // check word len
                word_len = wcslen(current_word);
                if (word_len + 1 > max_word_len){
                    // решаем сколько памяти надо выделить
                    difference = word_len + 1 - max_word_len;
                    if (difference > max_word_len){
                        next_max_word_len = difference + max_word_len;
                    } else {
                        next_max_word_len = max_word_len * 2;
                    }
```

```
                    max_word_len = next_max_word_len;
                    for (w = num_words; w < max_words_array_len; w++){
                        p_for_word = realloc(words[w], sizeof(wchar_t) *
max_word_len);
                        if (p_for_word){
                            words[w] = p_for_word;
                        }
                    }
                }
                // write new word
                wcscpy(words[num_words], current_word);
                occuriences[num_words]++;
                num_words++;
            }
        }
    }
    // free memory
    for (i = num_words; i < max_words_array_len; i++){
        free(words[i]);
    }

    substrings->words = words;
    substrings->num_occur = occuriences;
    substrings->num_words = num_words;
}


int get_number_of_latin_letters(struct Word word){
    int i = 0;
    int number = 0;
    while (word.symbols[i] != L'\0'){
        if ((word.symbols[i] >= L'A' && word.symbols[i] <= L'Z') ||
(word.symbols[i] >= L'a' && word.symbols[i] <= L'z')){
            number += 1;
        }
        i += 1;
    }
    return number;
```

```
}

int compare_words(const void* a, const void* b){
    struct Word* word_1 = (struct Word*)a;
    struct Word* word_2 = (struct Word*)b;

    if (word_1->number_of_latin_letters < word_2-
>number_of_latin_letters){
        return 1;
    }
    if (word_1->number_of_latin_letters > word_2-
>number_of_latin_letters){
        return -1;
    }
    return 0;
}


void set_k(struct Text* text){
    int i;
    for (i = 0; i < text->text_len; i++){
        if (check_kirl(&(text->sentences[i]))){
            reverse(&(text->sentences[i]));
        }

    }
}
```

Файл string_functions.h

```
#ifndef SF
#define SF
#include "string_structures.h"
#include "substring_struct.h"

void get_substrings(struct Substrings* substrings, struct Text* text);
int compare_words(const void* a, const void* b);
int get_number_of_latin_letters(struct Word word);
void set_k(struct Text* text);
#endif
```

Файл sring_structures.h

```
#ifndef SS
#define SS
#include <wchar.h>

struct Word {
    wchar_t* symbols;
    int number_of_latin_letters;
};

struct Sentence {
    struct Word* words;
    wchar_t* chars;
    int words_number;
    wchar_t* separators;
};

struct Text {
    struct Sentence* sentences;
    int text_len;
};

#endif
```

Файл substring_struct.h

```
#ifndef SST
#define SST

// for task 1
struct Substrings {
    wchar_t** words;
    int* num_occur;
    int num_words;
};

#endif
```