Федеральное государственное автономное образовательное учреждение высшего образования

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

**Факультет информатики, математики и компьютерных наук**

**КУРСОВАЯ РАБОТА**

Создание словаря для текстового документа

по направлению подготовки 09.03.04 "Программная инженерия"

образовательная программа «Программная инженерия»

Выполнил:

Студент группы 18ПИ2

Данилов Сергей Дмитриевич

Руководитель:

Лупанова Елена Александровна

Старший преподаватель

базовой кафедры компании «МЕРА»

Нижний Новгород 2018

**Оглавление**

1. Введение…………………………………………………………………….2
2. Постановка задачи………………………………………………………….3
3. Обзор аналогов……………………………………………………………..5
4. Разработка программы……………………………………………………..6
5. Тестирование программы………………………………………………...16
6. Заключение………………………………………………………………..18
7. Руководство пользователя………………………………………………..19
8. Список использованных источников ….……………………………….22
9. Приложение……………………………………………………………….23

**Введение**

В настоящее время существует огромный спрос на прикладное программное обеспечение. Одним из его видов является семейство анализаторов правописания. Потребность в проверки текстов на правильность написания присутствует во многих областях науки, управления, бизнеса, так как создание грамотной документации гарантирует успех во многих сферах. Необходимость написания грамотных текстов, различных документов прослеживается и в жизни каждого человека – например, правильно написанное обращение дает больший шанс устроиться на работу, грамотно составленный реферат может быть высоко оценен. Но в чем здесь могут помочь информационные технологии? Главная проблема состоит в том, что большие документы довольно сложно проверять «вручную». Не каждый человек знает правильное написание всех слов русского языка, поэтому для проверки нужно использовать большие орфографические словари, а для нахождения интересующего слова требуется некоторое время и усилия. А иногда человек может банально пропустить неправильно написанное слово из-за невнимательности, что делает стопроцентно точную проверку без подручных средств очень сложной. На основе этого мы видим, что проблема сложности проверки текста вручную весьма актуальна. Целью моей курсовой работы является решение поставленной проблемы с помощью создания программы – анализатора орфографии.

**Постановка задачи**

Моей задачей является создание программы, использующей заранее подготовленный словарь, функционалом которой является проверка текстового документа, написанного на русском языке, и помощь пользователю в замене ошибочных слов на наиболее подходящие варианты. Разработанная программа ясно бы иллюстрировала практическое применение словаря в электронном виде. Чтобы решить эту задачу, нужно выполнить несколько подзадач. Прежде всего, необходимо ознакомиться с существующими решениями, оценить их и сделать выводы, далее изучить основные библиотеки си для работы с консолью, файловой системой, текстом и памятью. После необходимо изучить базовые алгоритмы работы с данными в области решаемой задачи. На основе полученных данных мной должен быть разработан алгоритм, позволяющий определять с помощью словаря, ошибочно данное на вход слово или нет. Если правописание нарушено, то, опираясь на словарь, алгоритм должен выдавать наиболее подходящие для замены варианты слов, или показывать, что вариантов нет. Этот алгоритм должен работать быстро, чтобы общее время работы алгоритма возрастало в зависимости от количества слов, подаваемых на вход, линейно. В дополнении к этому, алгоритм должен искать самые подходящие варианты исправления. На основе разработанной системы анализа нужно реализовать программу, которая на вход должна принимать текстовой документ формата “.txt” любого размера. Программа должна отобразить этот документ на экран и отдельно выделить слова с орфографическими ошибками. Пользователь должен решить, какие слова он изменит, какие оставит неизменным. Если конкретные слова нужно изменить, пользователь может сам исправить написание, а может выбрать один из нескольких вариантов, которые предоставит программа. Если пользователь завершил работу с текстом, он может сохранить изменения в этом файле или сохранить под новым именем. Также необходимо помнить, что программа базируется на электронном словаре в формате текстового документа “.txt”. Для его создания нужно найти такой список русских слов, который включал бы в себя большое количество (около 180000) слов на русском языке в различных формах, на основе его создать текстовый документ, после подготовив его для правильной работы программы. Вот полный список требований к реализуемой программе:

1. Высокая скорость загрузки и подготовки программы к работе (не более 20 секунд).
2. Высокая скорость анализа файла (слов в миллисекунду).
3. Точное определение верного и ошибочного написания (не более одной ошибки на 500 слов).
4. Приложение и необходимые для его работы файлы должны весить вместе не более 20 Мб.
5. Программа должна иметь консольный интерфейс и приятное оформление.
6. Управление функционалом должно быть простое и понятное.
7. Приложение должно быть совместимым с операционными системами Windows 95 и выше, должна использовать не более 100 Мб оперативной памяти.
8. Требования к аппаратной части: 512 Мб RAM, Intel Pentium и более современные версии, 20 Мб места на жестком диске.

Рассмотрим пример работы программы. Допустим, что программа использует электронный словарь, реализованный как текстовый документ *dictionary.txt*, в котором записаны три слова – *“мама”, ”мыла”, ”раму”*. Пусть на вход приложению подается текстовый документ *in.txt,* в котором содержатся три русских слова через пробел: “*мама маму соль*”. Программа последовательно извлекает эти слова из текста и проверяет их по словарю. Слово *“мама”* содержится в словаре, поэтому приложение с ним не работает. А вот слова *«маму»* в словаре нет, поэтому программа выбирает лучшие варианты замены. Слова *“мама” и “раму”* максимально похожи на первоначальное слово “маму”, потому что отличаются на одну букву. Слова *“соль”* также нет в словаре, но оно отличается тем, что для него нельзя найти наиболее подходящий вариант из списка. После содержимое файла выводится на экран, а ошибочные слова особо выделяются (“*мама маму соль*”.). Для слова *“маму”* пользователь может выбрать замену из найденных ранее вариантов, которые высвечиваются на экране, а может сам придумать для них замену. Для слова “соль” программа должна сообщить пользователю, что замен не найдено, и предложить пользователю самим исправить слово. После работы с документом пользователь может сохранить изменения в текущем файле, а может для этого создать новый файл, например “*out.txt*”. Результат может быть, например, таким:

* *“мама маму соль”* (пользователь ничего не изменил)
* *“мама* ***раму*** *соль”* (пользователь заменил слово *“маму”* на *“раму”,* которое было предложено программой)
* *“мама* ***мама моль****”* (пользователь заменил слово маму на мама, а соль на моль) и так далее.

Перед тем, как начать реализовывать идею, нужно выяснить, какие аналоги программ со схожей концепцией существуют на данный момент.

**Обзор существующих решений**

Проводя поиск аналогичных программ по анализу текстов, я пришел к выводу, что список их не очень большой. Из них я выделил три основных приложения – аналога моей программы. Это программа Word от компании Microsoft, программа ОРФО 2016 российской компании «Информатик» и веб-приложение Text.ru компании “Текст Медиа”.

1. Microsoft Word. (+высокая скорость и точность проверки – высокая цена (от 249$ до 439$), присутствуют неточности при проверке документов на русском языке)
2. ОРФО 2016 (+высокая скорость и более глубокая проверка, благодаря сложным лингвистическим алгоритмам – высокая цена (от 1980руб. до 4980руб.)
3. Text.ru (+высокая точность –нагроможденный интерфейс, необходимость подключения к интернету, невысокая скорость проверки больших документов)

Итак, чтобы моя программа имела наиболее лучший функционал, чем аналоги, моя программа должна иметь небольшой вес, достигать высокой скорости анализа документов, работать автономно и содержать простой и доступный интерфейс.

**Разработка программы**

Сначала нужно описать структуру нашего решения – распределить программу на блоки, где для каждого блока предписан свой функционал. Реализация подпрограмм каждого блока будет содержаться в файлах .c, а прототипы этих функций и необходимые константы в заголовочных файлах .h. Итак, стержень нашей программы – файл main.c, в нём содержится главная функция программы *int main(),* в которой создается окно приложения, а также вызываются различные подпрограммы, связанные с работой приложения. Следующий блок – это модуль работы со словарём. Важнейшие функции этой части программы – инициализация словаря, определение корректности написания переданного в качестве аргумента слова и нахождения с помощью словаря наиболее подходящих замен для ошибочно написанного слова. Для него отведен файл dictionary.h с объявлениями функций и dictionary.c с их реализацией. Самая объёмная часть программы – блок Interface с файлами Interface.h и Interface.c. В нем выполняется интерактивная часть приложения – взаимодействие с пользователем, вывод необходимой информации на экран, взаимодействие с блоком работы со словарем, работа с файлами. Отдельно можно выделить блок описания структур данных и функций для работы с ними – *Templates.h и Templates.c.* В результате получаем такую структуру нашего проекта:

* Main
  + main.c
    - int main(int, args) (главная функция, точка входа приложения)
* Interface
  + Interface.h
  + Interface.c
* Dictionary
  + Dictionary.h
  + Dictionary.c
* Templates
  + Templates.h
  + Templates.c

Отдельно необходимо упомянуть текстовый файл – словарь. В этом файле под названием “Russian.txt”, находящемся в файле проекта, построчно записаны слова на русском языке во всевозможных формах, общее количество которых 1 601 915. Данный словарь был загружен с открытого репозитория “*russian-words*” пользователя системы контроля версий GitHub под ником *danakt*. Этот словарь создан благодаря анализу многочисленных электронных книг, статей, в результате которого отобраны слова на русском языке, а далее благодаря особым алгоритмам лингвистики и морфемики, было создано большое количество всевозможных форм этих слов. Примерный алгоритм таков – из слова извлекается корень, а после этого к нему добавляются различные морфемы, далее происходит поиск употребления получившихся слов в различных ресурсах. Если слово употребляется часто – оно добавляется в словарь. Если применить данную методику, получается набор различных слов, склоненных по лицам, числам, падежам, не включая имена собственные. Аналогичный словарь был создан пользователем CVS GitHub, и его мы будем использовать в решении нашей задачи.

Также внимания заслуживает тот факт, что программа работает с текстовой информацией на русском языке, для корректного ввода и отображении текста используется библиотека locale.h.

Теперь подробно разберем реализацию всех частей программы и их взаимодействие. Начнем со вспомогательного модуля Templates, потому что его возможности пригодятся во всех остальных модулях.

* Templates.h > *struct list*

*struct list {*

*struct list\* next;*

*struct list\* prev;*

*char\* string;*

*struct dblist\* strerrwrds;*

*};*

Данная структура данных – двусвязанный список с некоторыми особенностями. С помощью этого списка будет структурироваться исходный файл. В переменной string находится очередная строчка из текста, в next и prev ссылки на предыдущий и следующий элемент списка, в strerrwrds – ссылка на список, содержащий неправильные слова в этой строчке.

* Templates.h > *struct dblist*

*struct dblist {*

*struct dblist\* prev;*

*struct dblist\* next;*

*char\* errwrd;*

*};*

В таком двусвязном списке будут храниться неправильные слова из той строки, элемент списка которой содержит ссылку на него. Само неправильное слово можно получить по ссылке *errwrd.* На данной схеме изображена получившаяся составная структура данных:

*next*

*prev*

*string*

*strerrwrds*

*next*

*prev*

*errwrd*

* Templates.c > *struct list\** ***initlist****(char\* string);*

Создает двусвязный список list, единственный элемент которого содержит строку string. Возвращается ссылка на созданный список.

* Templates.c > *struct dblist\** ***initdblist****(char\* string);*

Создает двусвязный список dblist, единственный элемент которого содержит строку string. Возвращается ссылка на созданный список.

* Templates.c > *struct list****\* add2list****(struct list\* phead, char\* string);*

На вход принимает ссылку на «голову» списка и строку, необходимую добавить в структуру типа list. Проходя через весь список, функция ищет последний элемент, выделяет память под новый элемент, заполняет его свойства и обновляет ссылки. Возвращает ссылку на новый добавленный элемент (так сделано для удобства).

* Templates.c > *void* ***add2dblist****(struct list\* phead, char\* string);*

Алгоритм аналогичен предыдущему, только работает он со структурой dblist, в которую добавляется элементы с ссылками на неправильные слова. Функция ничего не возвращает.

* Templates.c > *void* ***next****(struct list\*\* ptmpstr, struct dblist\*\* ptmperr, struct list\* strings)*

В ходе программы нам нужно будет обходить все неправильные слова. Данная функция позволяет перейти к следующему неправильному слову в составной структуре данных. Она принимает указатель на текущий элемент в списке строчек, текущий элемент в списке неправильных слов и указатель на голову двухсвязного списка строчек. Алгоритм таков: если в переданном по ссылке ptmperr элементе свойство next = NULL, функция изменяет указатель по ссылке на элемент на строчку на следующий элемент со строчкой, а ptmperr ссылается на errwrds этого нового элемента. Иными словами – если в строчке нет следующего неправильного слова, программа меняет текущую строку на следующую, а текущее неправильное слово на первое в новой строчке. Если же в исходной строке еще имеются неправильные слова, указатель на текущую строку остается неизменным, а на текущее неправильное слово ссылка копируется из \*(ptmperr) -> next. Если же и строчки, и неправильные слова закончились, то с помощью ссылки string подпрограмма назначает текущей строкой первую в списке, а текущим неправильным словом – первое неправильное слово в этой строке. Этот и следующий алгоритмы нужны для перехода между неправильными словами для последующей работы с ними.

* Templates.c > *void* ***prev****(struct list\*\* ptmpstr, struct dblist\*\* ptmperr, struct list\* strings)*

Алгоритм аналогичен предыдущему, только здесь идет переход на предыдущие элементы.

* Dictionary.c > *void* ***fInit****(char\*\*\* dictionary, int\* dictionarylens)*

Функция загружает словарь из текстового документа russian.txt в память. В этом документе построчно записано … слов на русском языке в правильном написании. На вход принимается ссылка на переменную, в которой лежит массив на 37 элементов типа char\*\*, в него и будет загружен словарь. Для всех 37 элементов выделяется память на определенное количество ссылок типа char\*, это количество зависит от того, сколько слов данной длины содержится в словаре и будет загружено в память. Далее построчно считываются слова из словаря и распределяются в динамический массив. Нужно оговорить, что такой способ требует большого объема памяти, но этим достигается высокая скорость. Получаем такой массив:

* Dictionary.c > *char\** ***fCheck****(char\* word, char\*\*\* dictionary, int\* dictionarylens)*

Данная функция проверяет переданное по ссылке word слово на правильность написания по словарю, ссылку на который мы передаем в dictionary, а вспомогательный массив, упоминаемый ранее через ссылку dictionarylens. Предварительно слово проверяется на то, что в нем нет цифр, затем регистр всех символов меняется на нижний. Далее это слово сравнивается со всеми словами, схожими по длине, и, в случае успеха, возвращается ссылка на аналогичное заданному слово в словаре. (Код функции проиллюстрирован в приложении)

* Dictionary.c > *struct result* ***fFind****(char\* word, char\*\*\* dictionary, int\* dictionarylen)*

Данная функция находит максимально похожие слова из словаря для замены неправильно написанного слова по ссылке word. Что касается алгоритма, то у меня был выбор между двумя возможными путями решения данной задачи. Один из низ – это последовательное сравнение всех символов данной длины в словаре. Этот способ довольно быстрый, но не имеет особой практической пользы. Этого нельзя сказать про второй способ – алгоритм известного разработчика и исследователя, сотрудника компании Google Питера Норвига. Он заключается в создании массива строк (путем различного выделения подстрок, фрагментации и т.д., осуществляемые методами библиотеки *string.h)*, в котором рассматриваются возможные ошибки, а именно

1. Вставка лишней буквы (массив с удалением букв из слова)
2. Пропуск буквы (массив со вставкой букв в слово)
3. Неправильное написание буквы в слове (массив с заменами букв в слове)
4. Ошибка в порядке букв в слове (массив со всевозможными заменами порядка букв в слове)

Далее из этого массива с помощью функции fCheck выбираются те, которые присутствуют в словаре, и как раз они будут подходящими вариантами замены ошибочного слова. Например, есть неправильно написанное слово “нуф”. Создаются слова: [“ну”, “нф”, “уф”, “нут”, “нул”, “науф”, …] . Далее эти слова ищутся в словаре и в случае успеха заносятся в список результата. В нашем случае слово “нут” есть в словаре, оно и будет предложено. Существенным недостатком данного алгоритма является его скорость, но он позволяет более глубоко проанализировать слово и предложить наиболее точную замену. Естественно, мне важно именно практическое применение продукта, и поэтому я выбрал второй вариант. В итоге возвращается структура типа struct result, полями которой являются переменные *char\*\* arr*, содержащая ссылку на динамический массив с указателями на строки, наиболее подходящими на исправление, и количество таких слов– *int length*. (Код функции проиллюстрирован в приложении)

* *Interface.c > FILE\** ***fOpenfile****(char\* arg, char\*\* fname\_out)*

С помощью этой функции открывается входной файл. В качестве аргумента передается аргумент командной строки (он может быть NULL), а также ссылка на переменную, куда запишется имя файла (он нужен будет при окончании работы с файлом). Если имя файла не было передано в качестве аргумента командной строки, программа просит пользователя ввести его в окне приложения. Если произошла ошибка, программа попросит повторить ввод. Функция возвращает указатель на открытый файл FILE\*.

* Interface.c > *struct list\** ***fMainProc****(FILE\* ned, char\*\*\* dictionary, int\* dictionarylens)*

Такое название функция имеет из-за того, что здесь выполняется самая важна часть программы – исходный файл разбивается на строчки и в них определяются слова с ошибками. Сначала создается структура struct list – двусвязный список, описанный ранее. Далее из файла поочередно считываются все строчки. Каждую итерацию добавляется элемент, где поле string ссылается на считанную строку, а для поля errwrds создается список struct dblist. Данная строчка разбивается на слова функцией strtok и каждое слово проверяется на корректность написания функцией модуля Dictionary fCheck. В случае нахождения ошибки в слове, оно добавляется в список неправильных слов. Такая процедура повторяется для всех строчек, и готовый список передается в качестве возвращаемого значения по ссылке. На данной схеме изображен возможный в результате работы подпрограммы список.

* Interface.c > *void* ***fOutput*** *(HANDLE hSTDout, struct list\* strings, struct dblist\* current)*

Данная функция выводит текст на экран и выделяет ошибки. На вход подается графический дескриптор приложения, определенный в библиотеке *winows.h* (необходим для изменения цвета шрифта и выделения), ссылка на двусвязный список со строчками и указатель на элемент типа struct dblist с неправильным словом, который на данный момент является текущим. Алгоритм таков: строчки посимвольно выводятся на экран методами стандартной библиотеки *stdio.h*, при этом ищется вхождение первого слова из списка неправильных в этой строке, и если очередной символ строки совпадает с вхождением неправильной строки на экран, то он печатается желтым шрифтом. Если это неправильное слово является еще и текущим выделенным, то оно выделяется красным цветом. Так обрабатываются все строчки в списке до конца.

* Interface.c > *char\** ***fChoose****(char\* wrd, char\*\*\* dictionary, int\* dictionarylens)*

Здесь выводится окно выбора замены для выбранного неправильного слова. В результате работы подпрограммы fFind мы получили массив возможных замен, его то мы и используем в этой функции. На экран под порядковыми номерами высвечиваются возможные варианты замены: под номером 0. – отмена, при выборе этого пункта приложение возвращается к обычному режиму работы. Пункт 1. – «свой вариант», при выборе этого варианта пользователь должен сам ввести слово, на которое будет заменена ошибка. Далее из массива выводятся наиболее подходящие для замены варианты под порядковыми номерами. Пользователь может выбрать вариант действий, введя номер пункта. В итоге возвращается слово, выбранное на замену, либо если пользователь решил отменить выбор, возвращается NULL.

* Interface.c > *void* ***fChange****(char\* wrd, struct dblist\* mistake, struct list\* string)*

В этой функции происходит замена неправильного слова (ссылка на него содержится в struct dblist\* mistake, а на элемент, содержащий строку с ошибкой - struct list\* string) на выбранное в предыдущем пункте слово по ссылке wrd. Для начала функция посимвольно проходит строку с ошибкой, чтобы найти вхождение ошибочного слова. Далее эта строка перезаписывается так, что найденное ошибочное слово меняется на выбранное слово по ссылке wrd средствами библиотеки string.h. При этом из списка неправильных слов errwrds элемент с этим словом удаляется, ссылки обновляются. В результате мы получаем исправленную строку. Стоит упомянуть, что функция ничего не возвращает.

* Interface.c > *FILE\** ***fOpenfilew****(char\* arg1, char\* arg2)*

Функция открывает файл на запись. На вход подаются arg1 – имя входного файла (его можно получить после работы функции fOpenFile), arg2 – второй аргумент командной строки (может быть NULL, тогда пользователь вручную вводит имя файла). Алгоритм аналогичен функции открытия файла на запись fOpenFile, но есть одно отличие: пользователь может нажать Q, и тогда исходный файл просто перезапишется. Возвращает указатель на открытый на запись файл.

* Interface.c > *void* ***fWrite\_in\_file****(FILE\* fout, struct list\* strings)*

Эта подпрограмма записывает в уже открытый на запись файл fout значения поля string всех элементов списка под указателем strings.

* Interface.c > *void* ***fKey*** *(char key)*

В данной функции обрабатываются нажатия клавиш пользователем во время выполнения основного цикла в функции main, перехваченные функцией getchar(). Ниже описаны действия каждой из клавиш:

1. Клавиша «вправо» (код ASCII 77):

Переходит к следующему неправильному слову. Для этого изменяет ссылку на текущее неправильное слово методом next и обновляет информацию на экране, подсвечивая следующее неправильное слово.

1. Клавиша «влево» (код ASCII 75):

Переходит к предыдущему неправильному слову. Для этого изменяет ссылку на текущее неправильное слово методом prev и обновляет информацию на экране, подсвечивая предыдущее неправильное слово.

1. Клавиша «Enter» (код ASCII 13):

Программа отчищает экран, выводит меню выбора замены для выбранного неправильного слова функцией fChoose. После приложение заменяет слово в строке на выбранное и удаляет его из списка неправильных слов для этой строки функцией fChange. Далее выделяется следующее неправильное слово методом next и информация на экране обновляется функцией fOutput.

1. Клавиша «Esc» (код ASCII 27):

Программа отчищает экран, выводит окно выбора файла для записи результата работы программы методом fOpenFilew. Программа записывает в файл все строчки из списка методом fWrite\_in\_file и закрывает файл. В конце приложение переходит к обычному режиму работы.

И главная функция приложения–*int* ***main****(int argc, char\*\* argv) в файле main.c*

1. Во-первых, программа обрабатывает аргументы командной строки. Если первый после имени программы аргумент не пустой, он становится именем входного файла, а если есть еще один аргумент, то он становится именем файла на запись.
2. Функцией setlocale из библиотеки locale.h программа настраивается на работу с русским языком.
3. Методом библиотеки windows.h устанавливаем дескриптор вывода в консоли HANDLE hSTDout.
4. Инициализируем структуру, содержащюю информациию о курсоре и окне CONSOLE\_SCREEN\_BUFFER\_INFO SCRN\_INFO и устанавливаем цветовой режим «желтый шрифт по синему фону»
5. Вызываем функцию fOpenFile из модуля Interface и сохраняем полученную ссылку на файл в переменной
6. Загружаем словарь методом fInit блока Dictionary
7. Подготавливаем пустой двусвязный список strings, куда будут сохраняться строки из файла методом initlist(NULL) и заполняем его методом fMainProc блока Interface. В итоге получаем готовый список с определенными в тексте ошибками.
8. Устанавливаем текущей ошибкой первую ошибку первой строчки, в которой у нас существует ошибка. На элемент со строчкой с текущей ошибкой будет указывать ссылка ptmpstr (в начале равна strings), на элемент с ошибкой – ptmperr (strings->strerrwrds).
9. Методом fOutput блока Interface выводим строчки на экран, подсвечиваем ошибки и выделяя первую из них как текущую.
10. Закрываем файл fin методом fclose(FILE\*);
11. Открываем «вечный» цикл, в котором будут обрабатываться нажатые пользователем клавиши через функцию fKey. В зависимости от нажатой клавиши выполняется определенный процесс, как описано выше.

Что касается дизайна, я решил использовать стиль приложений DOS. В разработке дизайна были заимствованы детали таких программ, как Free Pascal или Borland C. В качестве основного цвета шрифта был выбран белый, а цвет фона синий. Для выделения неправильных слов был выбран желтый цвет, а для обозначения текущего неправильного слова (можно сказать курсора) был выбран белый шрифт с красным фоном букв. Получился достаточно приятный дизайн и интуитивно понятный интерфейс, а это очень важные критерии, которые необходимо соблюдать при создании приложения.

После разработки кода, компиляции средствами среды разработки Microsoft Visual Studio получили готовое приложение – *orpho.exe*. Далее необходимо обнаружить возможные в ходе эксплуатации программы ошибки, а также провести тесты с целью понять, соответствует ли приложение заданным ранее условиям.

**Тестирование программы**

Сначала протестируем программу на правильность определения ошибок. Для этого подготовим 3 различных файлов формата .txt с текстами на русском языке, причем заранее известно, сколько ошибок сделано в каждом из них.

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование файла | Количество ошибок | Ошибок найдено | Ошибок не найдено |
| Test1.txt | 25 | 21 | 4 |
| Test2.txt | 55 | 45 | 10 |
| Test3.txt | 0 | 30+ | 0 |

В первом и во втором случае ошибки были не найдены, потому что для некоторых слов правильное написание зависит от контекста, и его можно определить только с помощью специальных правил, а не с помощью словаря. Что касается третьего файла, то в нём находятся слова не на русском языке и имена собственные, поэтому слова такого типа программа сочла ошибочными. Не смотря на это, программа не маркирует ошибочные слова не так часто, что дает нам право сказать, что с основной задачей программа справляется довольно успешно.

Далее нужно проверить скорость работы нашей программы. Используем те же файлы и отладчик Visual Studio 2017 и сравним зависимость времени выполнения работы (от старта до окончания вывода на экран) от размера файла. Пустой файл обрабатывается за 8 секунд – это время загрузки самого приложения. Рассмотрим 4 тестовых файла, по построенной диаграмме мы получаем линейную зависимость между размером файла и скоростью.

|  |  |  |
| --- | --- | --- |
| Наименование файла | Размер файла, Кб | Скорость работы, сек. |
| Test1.txt | 0.5 | 8.5 |
| Test2.txt | 2.5 | 10 |
| Test3.txt | 12 | 16 |
| Test4.txt | 24.4 | 27 |

При тестировании тем же отладчиком использование оперативной памяти выяснилось, что оно почти одинаково для файлов небольшого размера – приблизительно 127 мегабайт, так как основная информация, хранящаяся в памяти и необходимая для работы – это словарь.

Для ошибок тестового документа программа предлагает достаточно адекватные варианты замены. Единственный минус – это время ожидания ответа на запрос – для одного слова нужно примерно от 9 до 14 секунд. В данной таблице представлена зависимость между выбранными для исправления словами, предлагаемыми вариантами исправления и временем выполнения подпрограммы:

|  |  |  |
| --- | --- | --- |
| **Слово** | **Варианты исправления** | **Скорость, с.** |
| Гер***а***изм | Гебраизм, *героизм* | 9 |
| Самоотверже***н***ость | *Самоотверженность* | 9 |
| Ед***е***нения | Единения | 14 |
| Особе***н***о | Особно, особен, *особенно*, особно | 10 |

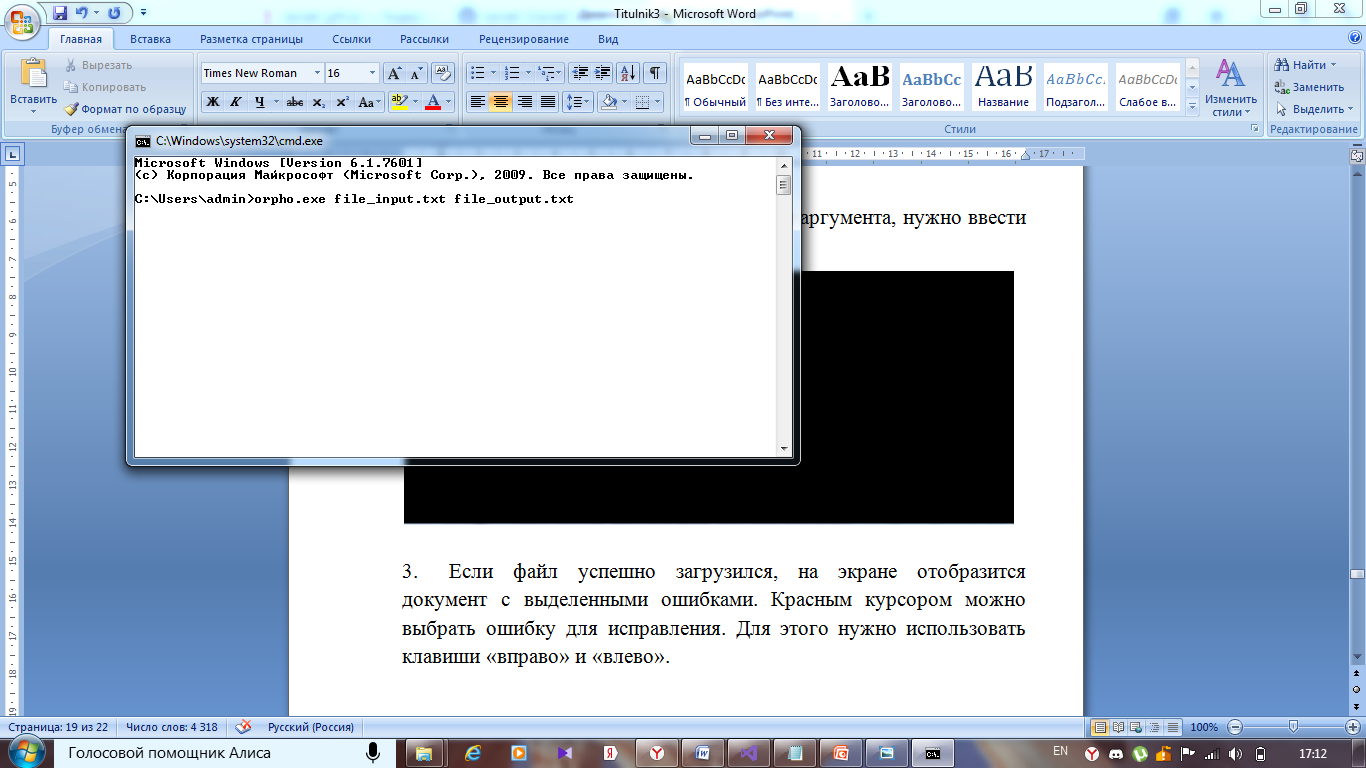
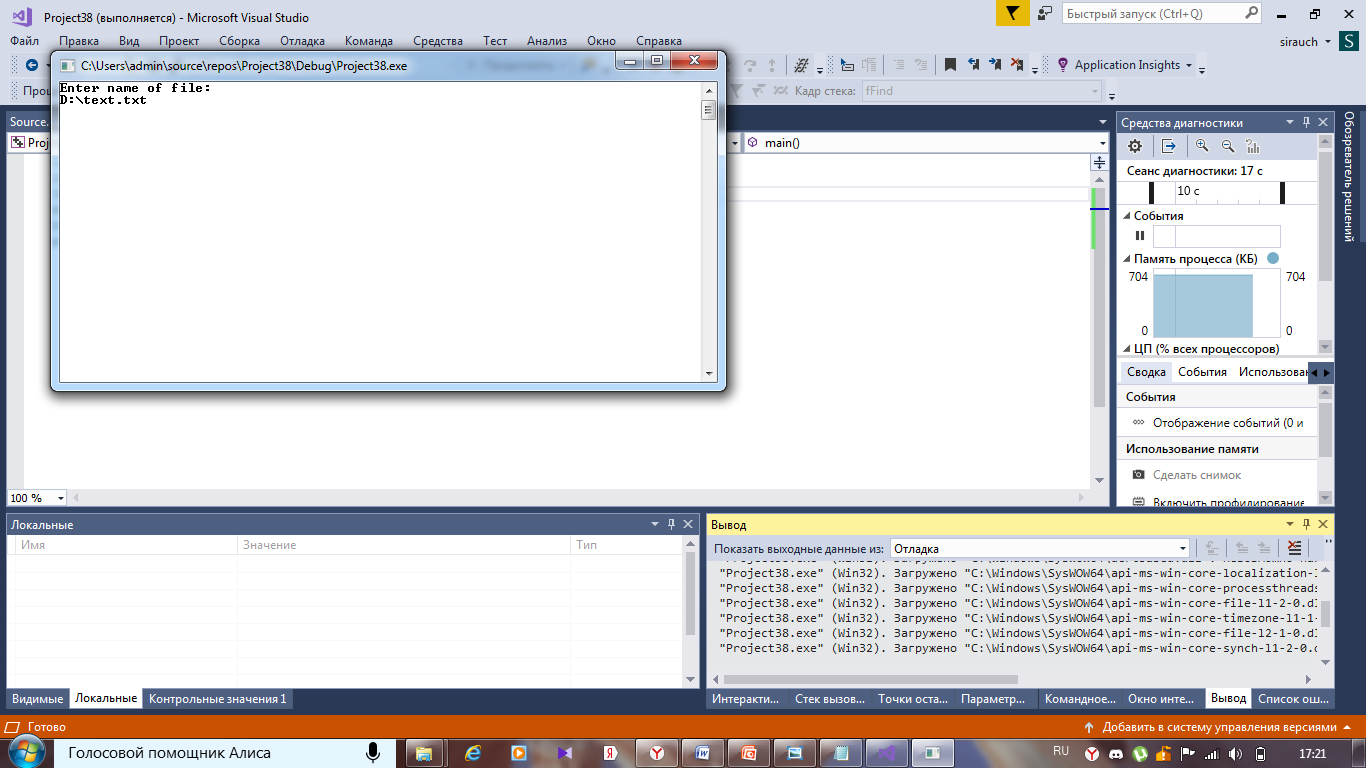
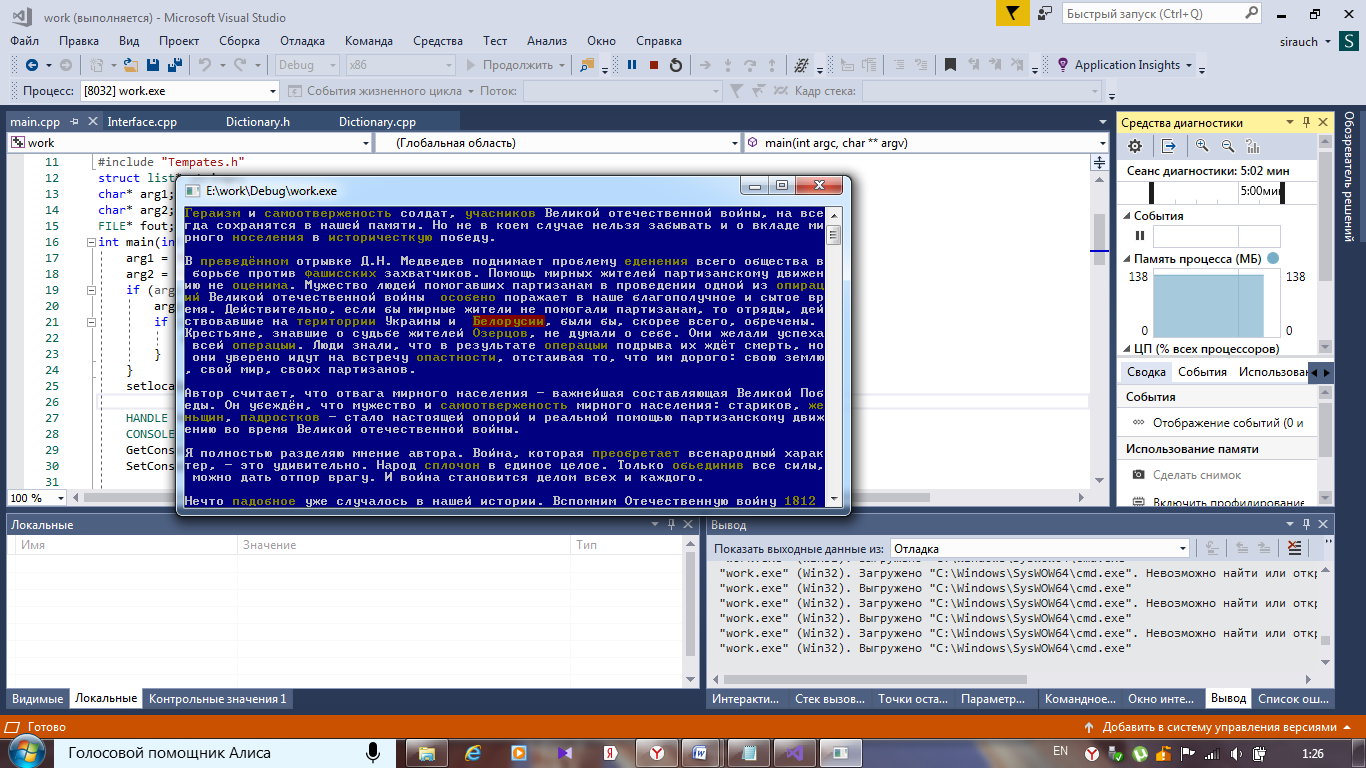
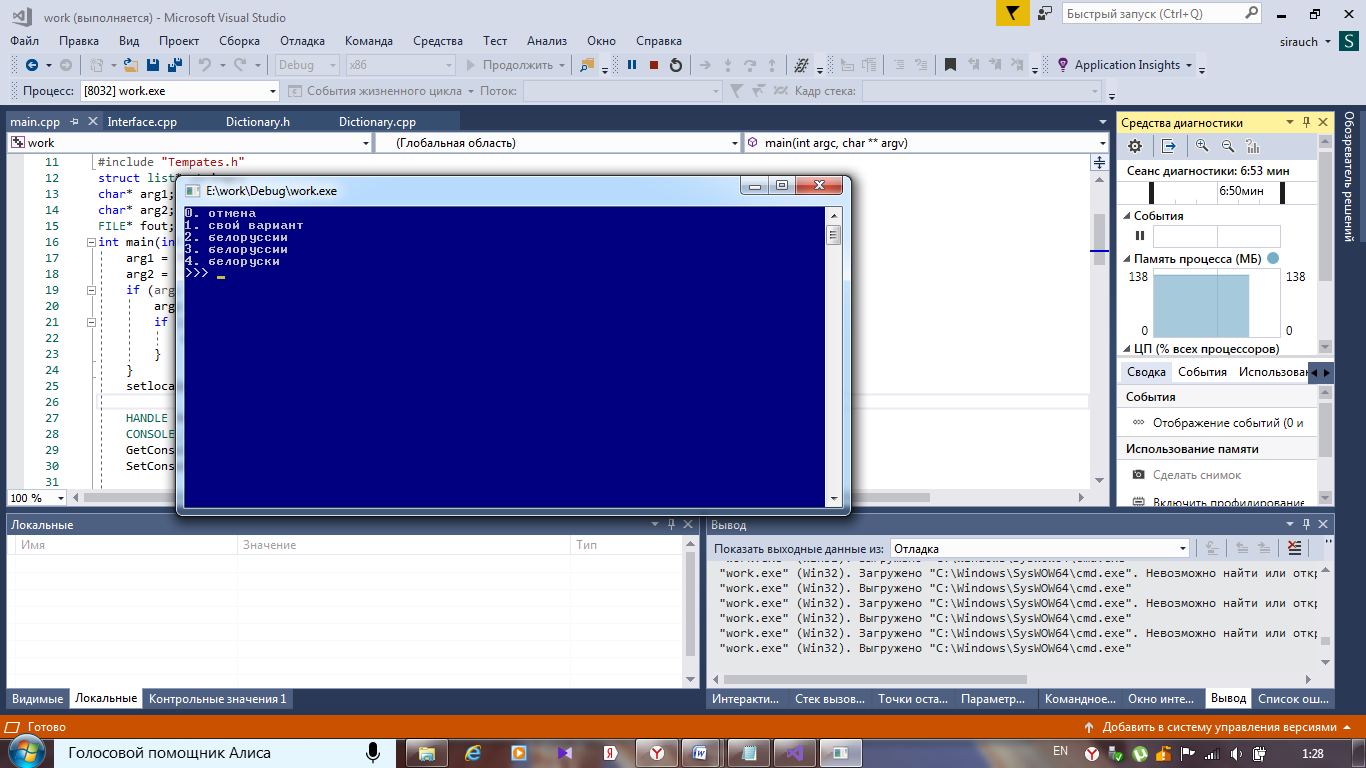
Рассмотрим отдельные сценарии работы программы. При работе с пустым документом ошибок не возникает, выводится просто пустой экран. При попытке открыть файл, имеющий формат не ‘.txt’, например, «*С:\file.txx*», отображается предупреждение, и программа просит повторно ввести имя файла. Приложение одинаково правильно работает и при открытии файла в самой программе, и при передаче имени файла в качестве аргумента командной строки. Если во время обработки попадается слово, которое содержит больше 37 букв, то оно автоматически маркируется как неправильное, потому что в словаре таких слов точно нет. Если во время работы алгоритма на вход подается строка, имеющая более 10000 символов, то она обрезается до допустимого количества символов. При вводе некорректного файла на запись результат работы программы, выводится предупреждение и предлагается повторить ввод.

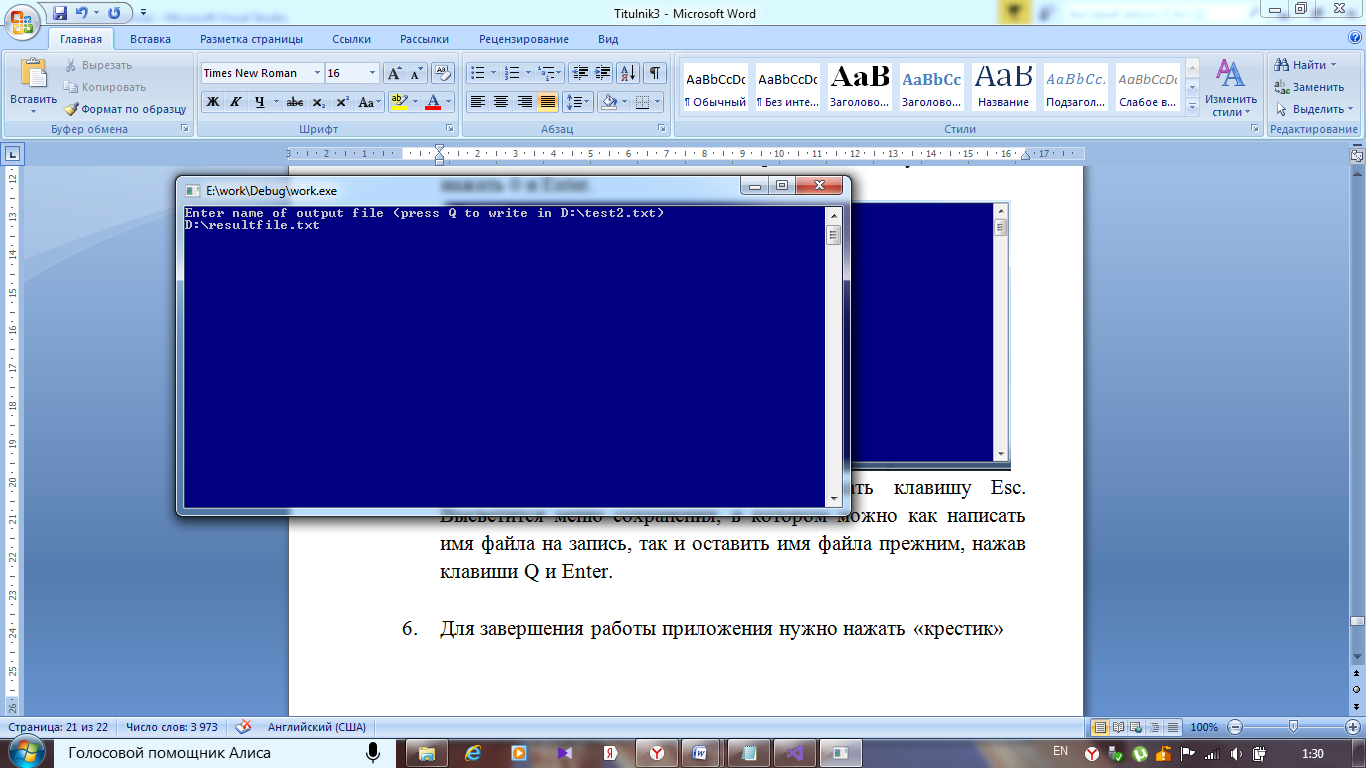
В итоге мы с уверенностью можем сказать, что программа удовлетворяет всем поставленным условиям по корректности, скорости программы, объему занимаемой оперативной памяти и объему программы и её файлов.

**Заключение**

Передо мной была поставлена цель - разработать программное обеспечение, главная цель которого – орфографическая проверка текста в электронном документе с возможностью замены ошибок, руководствуясь предоставляемыми программой подсказками. Дополнительно, созданное решение должно удовлетворять некоторым характеристикам, связанным со скоростью работы приложения, объемом занимаемой памяти и другим. Чтобы достичь этой цели, мне нужно было выполнить несколько задач – составить алгоритм проверки слов, алгоритм поиска наиболее похожих слов, продумать устройство программы, ее интерфейс и реализовать это с помощью языка программирования Си. После выполнения этих задач, я получил работоспособное приложение. Во время разработки программы, я приобрел опыт в создании структур данных, работой с текстовой информацией, файлами, приобрел опыт создания консольного интерфейса, узнал много нового о возможностях языка Си и библиотек из разных источников. В ходе тестирования этого приложения я сделал вывод, что моя программа полностью отвечает всем требованиям, предъявленным приложению, поэтому цель можно считать выполненной. Программа готова к использованию и может быть установлена всеми желающими. По моему мнению, более всего это приложение даст неоценимую помощь образовательным учреждениям, студентам, журналистам, всем, чья деятельность так или иначе связанна с копирайтингом. Но не смотря на полностью выполненное задание, мой проект нуждается в дальнейшем развитии, так как нужно создать более удобный дизайн, улучшить характеристики работы, такие как скорость, корректность работы. В частности, можно добавить современные лингвистические алгоритмы определения орфографических ошибок, связанных с невыполнением правил языка. Также нужно расширить возможности приложения, подключив поддержку других языков. В итоге, при дальнейшей поддержке данного решения может получиться достойный продукт.

**Руководство пользователя**

1. Чтобы начать работу с приложением, его необходимо запустить. Для этого можно использовать консоль, где в качестве аргументов можно указать имя файла для проверки. Также можно перетащить нужный файл на иконку с приложением.
2. Если файл не был передан в качестве аргумента, нужно ввести имя файла.
3. Если файл успешно загрузился, на экране отобразится документ с выделенными ошибками. Красным курсором можно выбрать ошибку для исправления. Для этого нужно использовать клавиши «вправо» и «влево».
4. Для исправления какой-либо ошибки выберите это слово и нажмите клавишу Enter. В высветившемся меню выберите наиболее подходящий вариант исправления, введя его номер и нажав Enter. Можно ввести свой вариант, тогда нужно ввести 1 и нажать Enter, а после написать свой вариант исправления. Если пользователь не хочет исправлять ошибку, он может нажать 0 и Enter.

Для сохранения изменений нужно нажать клавишу Esc. Высветится меню сохранения, в котором можно как написать имя файла на запись, так и оставить имя файла прежним, нажав клавиши Q и Enter.

1. Для завершения работы приложения нужно нажать «крестик»

**Список использованных источников**

1. Никлаус Вирт Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010 – 272 с.: ил.
2. Скиена С. Алгоритмы. Руководство по разработке – 2-е изд.: Пер. с англ. – СПб. БХВ-Петербург, 2011 – 720 с.; ил.
3. Такташкин Д.В., Мокроусова Е.А. Методы и алгоритмы проверки орфографии тестовых документов // Современные научные исследования и инновации. 2017. № 5 [Электронный ресурс]. URL: http://web.snauka.ru/issues/2017/05/72892 (дата обращения: 25.03.2019).
4. Абрамович А. Как у вас с грамотностью? Или несколько способов проверить текст на ошибки [Электронный ресурс]/ Режим доступа: https://artabr.ru/kak-u-vas-s-gramotnostyu/, свободный. (Дата обращения: 10.05.2019 г.).
5. Peter Norvig.How to Write a Spelling Corrector [Электронный ресурс]/ URL: <https://www.norvig.com/spell-correct.html> (Дата обращения: 8.05.2019 г.)
6. Danakt Frost. Список русских слов [Электронный ресурс]/ URL: <https://github.com/danakt/russian-words> (Дата обращения: 1.05.2019)

**Приложение**

*Код функции проверки слова на правильность написания*

char\* fCheck(char\* word, char\*\*\* dictionary, int\* dictionarylens) {

char wrds[37];

const char\* nums = "0123456789\0";

strcpy(wrds, word);

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

wrds[i] = tolower(wrds[i]);

for (int j = 0; j < 10; j++) {

if (wrds[i] == nums[j]) {

return 0;

}

}

}

for (int j = 0; j < \*(dictionarylens+strlen(wrds)); j++) {

if (strcmp(\*(\*(dictionary+strlen(wrds)) + j), wrds) == 0) {

return \*(\*(dictionary + strlen(wrds)) + j);

}

}

return 0;

}

*Код функции нахождения вариантов замены ошибочному слову*

struct result fFind(char\* word, char\*\*\* dictionary, int\* dictionarylen) {

int length = strlen(word);

char\*\* arrReturn = (char\*\*)malloc(sizeof(char\*) \* 100 \* length);

int iter = 0;

struct couple\* splits = (struct couple\*)malloc(sizeof(struct couple)\*(length + 1));

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

if (i == 0) { (splits)->first = (char\*)malloc(sizeof(char) \* 2); strcpy((splits)->first, ""); }

else { (splits + i)->first = (char\*)malloc(sizeof(char)\*(i + 1)); for (int j = 0; j < i; j++) { \*((splits + i)->first + j) = \*(word + j); } \*((splits + i)->first + i) = '\0'; }

if (i == length) { (splits + i)->second = (char\*)malloc(sizeof(char)); strcpy((splits + i)->second, " "); }

else { (splits + i)->second = (char\*)malloc(sizeof(char)\*(length - i + 1)); for (int j = 0; j < length - i; j++) { \*((splits + i)->second + j) = \*(word + i + j); } \*((splits + i)->second + (length - i)) = '\0'; }

}

char\* bufferA = (char\*)malloc(sizeof(char)\*length);

char\* bufferB = (char\*)malloc(sizeof(char)\*length);

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

strcpy(bufferA, (splits + i)->first);

strcpy(bufferB, (splits + i)->second);

if (strcmp(bufferB, "") != 0) { strcat(bufferA, bufferB + 1); }

char\* tmp;

if ((tmp = fCheck(bufferA, dictionary, dictionarylen)) != NULL) {

\*(arrReturn + iter) = tmp;

iter++;

}

}

char chars[35] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя”;

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

strcpy(bufferA, (splits + i)->first);

strcpy(bufferB, (splits + i)->second);

for (int j = 0; j < 35; j++) {

strcpy(bufferA, (splits + i)->first);

char insert[2];

insert[0] = \*(chars + j);

insert[1] = '\0';

strcat(bufferA, insert);

strcat(bufferA, bufferB);

char\* tmp;

if ((tmp = fCheck(bufferA, dictionary, dictionarylen)) != NULL) {

\*(arrReturn + iter) = tmp;

iter++;

}

}

}

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

strcpy(bufferA, (splits + i)->first);

strcpy(bufferB, (splits + i)->second);

for (int j = 0; j < 35; j++) {

if (strcmp(bufferB, "") != 0) {

strcpy(bufferA, (splits + i)->first);

char insert[2];

insert[0] = \*(chars + j);

insert[1] = '\0';

strcat(bufferA, insert);

strcat(bufferA, bufferB + 1);

char\* tmp;

if((tmp = fCheck(bufferA, dictionary, dictionarylen)) != NULL){

\*(arrReturn + iter) = tmp;

iter++;

}

}

}

}

return { arrReturn,iter };

}

Добрый день уважаемая комиссия, меня зовут Данилов Сергей, я студент 1 курса специальности Программная инженерия. Разрешите представить вашему вниманию проект «Создание словаря для текстового документа». (!) Я не случайно выбрал именно эту тему, так как существует стабильный спрос на инструменты для работы с текстами в таких областях, как наука, статистика. Итак, исследуя нашу область, необходимо обратиться к понятию словаря. (!)Как мы знаем, это книга, содержащая перечень слов, обычно с пояснениями, толкованиями или переводом на другой язык. Что это нам даёт? Если, например, в нем будет отображено корректное написание разных форм слов - используя этот список можно проверить наличие орфографических ошибок в большом количестве текстов. (!) Проводя проверку вручную, мы можем столкнуться с проблемой затруднительности использования данного метода. Именно тогда нам на помощь приходят прикладная информатика и программная инженерия. (!) Итак, цель данного проекта - создание программы, функционалом которой является проверка текстового документа, написанного на русском языке и помощь пользователю в замене ошибочных слов на наиболее подходящие корректные варианты. Данная программа иллюстрировала бы практическое применение созданных различными способами словарей. В частности, цель проекта можно разделить на несколько задач, таких как - Подробное изучение области проекта, создание алгоритма, при помощи приобретенного знания, позволяющего выполнить поставленную задачу, реализация идеи в виде программного продукта, тестирование получившегося решения с целью определить – отвечает ли созданное решение предъявленным критериям. А они выражаются так – Во-первых – созданное приложение должно иметь комфортный и понятный интерфейс, приятный дизайн, а во вторых должна отвечать отдельным системным характеристикам. Они кажутся достаточно большими, но они оправданны при том условии, что программа работает с огромным массивом данных, заложенным в словаре. Прежде чем приступать к реализации идеи, желательно проанализировать существующие решения в данной области, (!) такими как Microsoft word Орфо Text.ru + высокая точность, скорость проверки – высокая цена и излишний функционал для выбранной задачи. Из этого анализа можем сделать вывод: чтобы программа была успешно реализована, она должна быть легкая, простая, но эффективная. (!) Приступим к реализации нашего проекта. Начнем со стержня нашей системы – словаря, который содержал бы различные форм склонений русских слов, записанные в текстовом документе. (!) Такие списки создаются благодаря анализу огромного количества различных текстовых источников методами big data, нейросетей, применения специальных лингвистических алгоритмов, но к сожалению, у меня нет такого опыта и необходимых знаний в этой области, поэтому я воспользовался помощью такого полезного ресурса, как GitHub, (!) так, в открытом репозитории пользователя под ником Danakt Frost находится словарь – список русских слов – включающий около 1500000 слов. Имея его, становится возможным создать нужную нам систему. Итак, приступим к реализации алгоритма. (!) explanation (!) Используя возможности системы разработки VS и базируясь на возможностях языка Си, создадим программный продукт реализующий идею, работу получившейся программы мы можем увидеть на видео.(ВИДЕО) (!) После разработки необходимо протестировать программу, чтобы предотвратить ошибки или улучшить ее перед релизом. Проведя тестирование средствами отладчика VS, я пришел к выводу, что заданным ранее параметрам программа полностью соответствует. (!) В заключении я хочу сказать, что поставленная мной задача показать практическое применение создания словарей через создание программы проверки орфографии успешно выполнена. Вместе с этим я приобрел опыт в анализе источников при создании собственных идей, приобрел опыт разработки на языке Си, сам опыт сопровождения ПО.