Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Расчётно-графическая работа

по дисциплине: «Программирование»
Тема: «Разработка программы translate для перевода текста с помощью словаря»

Выполнил:
студент группы ИКС-433
Воробьёв Дмитрий Андреевич

Новосибирск
2025

1 Задание

Разработать программу translate, выполняющую перевод текста с помощью словаря. Команда translate принимает на вход 3 файла. Первый содержит исходный текст, который необходимо перевести. Второй файл имеет вид простейшего словаря, где каждому слову на исходном языке соответствует слово на целевом. Третий файл необходимо создать и записать в него результат работы переводчика. Формат исходного текста должен быть сохранен.

2 Анализ задачи

2.1 Общий план работы программы

Программа выполняет следующие шаги:

1. Парсинг аргументов командной строки
2. Загрузка словаря в хеш-таблицу
3. Определение типа входных данных (файл/директория)
4. Обработка текста с учетом регистра и пунктуации
5. Перевод слов с использованием словаря
6. Поиск перевода в интернете (если разрешено)
7. Сохранение результата в указанный файл/директорию

2.2 Хеш-таблица

Для хранения словаря используется хеш-таблица с методом цепочек для разрешения коллизий. Основные характеристики:

- Хеш-функция: djb2
- Размер таблицы: 100 элементов
- Структура узла: ключ, значение, указатель на следующий узел

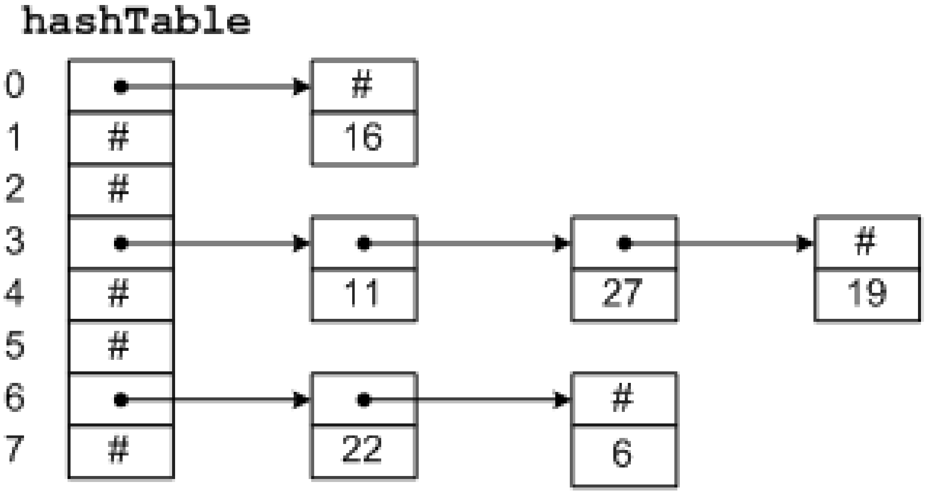


Рис. 1: Схема хеш-таблицы с методом цепочек

2.3 API для перевода

Для онлайн-перевода используется MyMemory API (`api.mymemory.translated.net`) с следующими параметрами:

- Метод: GET
- Параметры: q (текст), langpair (языковая пара)
- Ответ: JSON с полем responseData.translatedText

3 Тестовые данные

3.1 –help

```
$ ./translate --help
Использование: translate [ОПЦИИ] ВХОДНОЙ_ФАЙЛ СЛОВАРЬ ВЫХОДНОЙ_ФАЙЛ
Опции:
  -y              Автоматически подтверждать все запросы
  -n              Не подтверждать запросы автоматически
  --no-internet   Отключить поиск в интернете
  -e EXT          Указать расширение файлов для обработки (по умолчанию: txt)
  -t THREADS      Указать количество потоков (по умолчанию: 1)
  -h, --help      Показать эту справку
```

3.2 Корректные данные

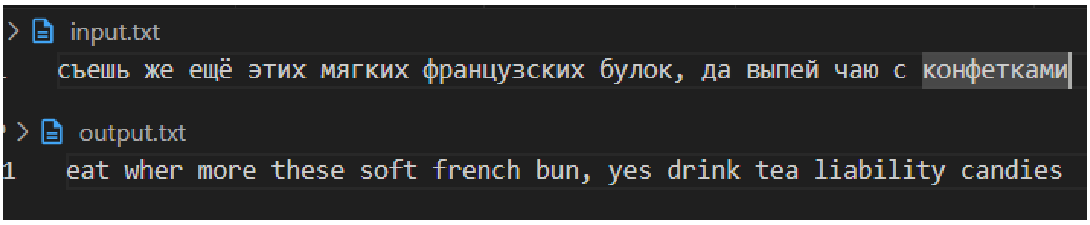
```
$ ./translate input.txt dictionary.txt output.txt
Файл output.txt уже существует. Перезаписать? (y/n): y
Слово 'этих' не найдено в словаре. Искать в интернете? (y/n): y
Слово 'мягких' не найдено в словаре. Искать в интернете? (y/n): y
```

3.3 Некорректные данные

```
$ ./translate
Ошибка: Недостаточно аргументов
Использование: translate [ОПЦИИ] ВХОДНОЙ_ФАЙЛ СЛОВАРЬ ВЫХОДНОЙ_ФАЙЛ
Опции:
  -y                Автоматически подтверждать все запросы
  -n                Не подтверждать запросы автоматически
  --no-internet     Отключить поиск в интернете
  -e EXT            Указать расширение файлов для обработки (по умолчанию: txt)
  -t THREADS        Указать количество потоков (по умолчанию: 1)
  -h, --help        Показать эту справку

$ ./translate -t 0 input_files dictionary.txt output_files
Ошибка: Количество потоков должно быть положительным числом
```

4 Скриншоты с результатами



```
съешь же ещё этих мягких французских булок, да выпей чаю с конфетками
```

```
eat wher more these soft french bun, yes drink tea liability candies
```

Рис. 2: Результат работы программы

5 Листинг программы

```
====== FILE: ./build.sh ======
mkdir Build
cd Build
cmake ..
make
cp translate ../translate
====== FILE: ./CMakeLists.txt ======
cmake_minimum_required(VERSION 3.10)
project(translator)

add_compile_options(-Wall -pedantic -finput-charset=UTF-8 -fexec-
    red↪ charset=UTF-8)

find_package(CURL REQUIRED)

find_path(JANSSON_INCLUDE_DIR jansson.h)
find_library(JANSSON_LIBRARY NAMES jansson)
```

```
if(JANSSON_INCLUDE_DIR AND JANSSON_LIBRARY)
    set(JANSSON_FOUND TRUE)
    set(JANSSON_INCLUDE_DIRS ${JANSSON_INCLUDE_DIR})
    set(JANSSON_LIBRARIES ${JANSSON_LIBRARY})
endif()

add_executable(translate
    main.c
    utils/hash_table.c
    utils/args_processing.c
    utils/file_processing.c
    utils/translation.c
    utils/internet_search.c
)

target_include_directories(translate PRIVATE ${
   red↪ CMAKE_CURRENT_SOURCE_DIR} ${JANSSON_INCLUDE_DIRS})

target_link_libraries(translate ${CURL_LIBRARIES} ${JANSSON_LIBRARIES
   red↪ } pthread)

find_library(CUNIT_LIBRARY NAMES cunit)
add_executable(test_hash_table
    tests/test_hash_table.c
    utils/hash_table.c
    utils/args_processing.c
    utils/file_processing.c
    utils/translation.c
    utils/internet_search.c
)

target_include_directories(test_hash_table PRIVATE ${
   red↪ CMAKE_CURRENT_SOURCE_DIR} ${JANSSON_INCLUDE_DIRS})

target_link_libraries(test_hash_table ${CUNIT_LIBRARY} ${
   red↪ CURL_LIBRARIES} ${JANSSON_LIBRARIES} pthread)

add_executable(test_translation
    tests/test_translation.c
    utils/hash_table.c
    utils/args_processing.c
    utils/file_processing.c
    utils/translation.c
    utils/internet_search.c
)

target_include_directories(test_translation PRIVATE ${
   red↪ CMAKE_CURRENT_SOURCE_DIR} ${JANSSON_INCLUDE_DIRS})
```

```
target_link_libraries(test_translation ${CUNIT_LIBRARY} ${
   red↪ CURL_LIBRARIES} ${JANSSON_LIBRARIES} pthread)

enable_testing()
add_test(NAME hash_table_test COMMAND test_hash_table)
add_test(NAME translation_test COMMAND test_translation)
===== FILE: ./main.c =====
#include "translate.h"

int main(int argc, char *argv[]) {
    ProgramOptions options = parse_arguments(argc, argv);

    if (options.input_path == NULL || options.dict_path == NULL ||
       red↪ options.output_path == NULL) {
        print_help();
        return 1;
    }

    HashTable *dictionary = create_hash_table(100);
    FILE *dict_file = fopen(options.dict_path, "r");
    if (dict_file == NULL) {
        printf("              :␣    ␣                ␣               ␣
           red↪          ␣               \n");
        return 1;
    }

    char line[256];
    while (fgets(line, sizeof(line), dict_file)) {
        char *key = strtok(line, "␣-");
        char *value = strtok(NULL, "␣-\n");
        if (key && value) {
            hash_table_insert(dictionary, key, value);
        }
    }
    fclose(dict_file);

    struct stat path_stat;
    stat(options.input_path, &path_stat);

    if (S_ISDIR(path_stat.st_mode)) {
        process_directory(options.input_path, options.output_path,
           red↪ dictionary, &options);
    } else {
        process_file_translation(options.input_path, options.
           red↪ output_path, dictionary, &options);
    }

    free_hash_table(dictionary);
    return 0;
}
===== FILE: ./requirements.sh =====
sudo apt update
sudo apt install libjansson-dev #
sudo apt install curl libcurl4-openssl-dev # http
===== FILE: ./start_tests.sh =====
./build.sh
cd Build
# ctest -V --output-on-failure
 ctest --output-on-failure
===== FILE: ./tests/test_hash_table.c =====
#include <CUnit/CUnit.h>
#include <CUnit/Basic.h>
#include "../translate.h"

void test_hash_table_operations(void) {
    HashTable* table = create_hash_table(100);

    hash_table_insert(table, "key1", "value1");
    hash_table_insert(table, "key2", "value2");

    CU_ASSERT_STRING_EQUAL(hash_table_search(table, "key1"), "value1"
       red↪ );
    CU_ASSERT_STRING_EQUAL(hash_table_search(table, "key2"), "value2"
       red↪ );
    CU_ASSERT_PTR_NULL(hash_table_search(table, "nonexistent"));

    hash_table_insert(table, "key1", "newvalue");
    CU_ASSERT_STRING_EQUAL(hash_table_search(table, "key1"), "
       red↪ newvalue");

    free_hash_table(table);
}

int main() {
    CU_initialize_registry();

    CU_pSuite suite = CU_add_suite("HashTable␣Tests", NULL, NULL);
    CU_add_test(suite, "hash_table␣operations␣test",
       red↪ test_hash_table_operations);

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
===== FILE: ./tests/test_translation.c =====
#include <CUnit/CUnit.h>
#include <CUnit/Basic.h>
#include "../translate.h"

void test_translate_word(void) {
    HashTable* dict = create_hash_table(100);
    hash_table_insert(dict, "cat", "          ");

    ProgramOptions options = {
        .no_internet = true,
        .auto_approve = false,
        .no_overwrite = false
    };

    char* trans1 = translate_word("cat", dict, &options, NULL);
    CU_ASSERT_STRING_EQUAL(trans1, "          ");
    free(trans1);

    char* trans2 = translate_word("unknown", dict, &options, NULL);
    CU_ASSERT_STRING_EQUAL(trans2, "unknown");
    free(trans2);

    free_hash_table(dict);
}

int main() {
    CU_initialize_registry();

    CU_pSuite suite = CU_add_suite("Translation␣Tests", NULL, NULL);
    CU_add_test(suite, "translate_word␣test", test_translate_word);

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
===== FILE: ./translate.h =====
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
#include <dirent.h>
#include <pthread.h>
#include <curl/curl.h>
#include <jansson.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
#include <wchar.h>
#include <wctype.h>
#include <getopt.h>

typedef struct HashNode {
    char *key;
    char *value;
    struct HashNode *next;
} HashNode;

typedef struct {
    HashNode **nodes;
    size_t size;
} HashTable;

typedef struct {
    char *input_path;
    char *dict_path;
    char *output_path;
    bool overwrite;
    bool no_overwrite;
    bool auto_approve;
    bool no_internet;
    char *file_extension;
    int thread_count;
} ProgramOptions;

typedef struct {
    char *input_file;
    char *output_file;
    HashTable *dictionary;
    ProgramOptions *options;
} ThreadData;

HashTable *create_hash_table(size_t size);
void free_hash_table(HashTable *table);
void hash_table_insert(HashTable *table, const char *key, const char
   red↪ *value);
char *hash_table_search(HashTable *table, const char *key);

ProgramOptions parse_arguments(int argc, char *argv[]);
void print_help();

void process_file_translation(const char *input_file, const char *
   red↪ output_file, HashTable *dictionary, ProgramOptions *options);
void process_directory(const char *input_dir, const char *output_dir,
   red↪  HashTable *dictionary, ProgramOptions *options);

char *search_translation_online(const char *word, const char *
    red↪ source_lang, const char *target_lang);
void add_word_to_dictionary(const char *word, const char *translation
    red↪ , HashTable *dictionary, const char *dict_path);

char *translate_word(const char *word, HashTable *dictionary,
    red↪ ProgramOptions *options, const char *dict_path);
char *translate_text(const char *text, HashTable *dictionary,
    red↪ ProgramOptions *options, const char *dict_path);
===== FILE: ./utils/args_processing.c =====
#include "args_processing.h"

void print_help() {
    printf("                              :␣translate␣[            ]␣
        red↪              _              ␣                 ␣
        red↪                  _                \n");
    printf("            :\n");
    printf("␣␣-y␣␣␣␣␣␣␣␣␣␣␣␣␣                              ␣
        red↪                            ␣        ␣                 \n");
    printf("␣␣-n␣␣␣␣␣␣␣␣␣␣␣␣␣      ␣                             ␣
        red↪                 ␣                              \n");
    printf("␣␣--no-internet␣␣␣                      ␣            ␣   ␣
        red↪                       \n");
    printf("␣␣-e␣EXT␣␣␣␣␣␣␣␣␣␣                 ␣                          ␣
        red↪                ␣        ␣                      ␣(       ␣
        red↪                       :␣txt)\n");
    printf("␣␣-t␣THREADS␣␣␣␣␣␣                 ␣                          ␣
        red↪                  ␣(      ␣                      :␣1)\n");
    printf("␣␣-h,␣--help␣␣␣␣␣␣                 ␣        ␣                   \
        red↪ n");
}

ProgramOptions parse_arguments(int argc, char *argv[]) {
    ProgramOptions options = {
        .input_path = NULL,
        .dict_path = NULL,
        .output_path = NULL,
        .overwrite = false,
        .no_overwrite = false,
        .auto_approve = false,
        .no_internet = false,
        .file_extension = "txt",
        .thread_count = 1
    };

    static struct option long_options[] = {
        {"no-internet", no_argument, 0, 0},
        {"help", no_argument, 0, 'h'},
        {0, 0, 0, 0}
    };
```

```
int opt;
while ((opt = getopt_long(argc, argv, "yne:t:h", long_options,
   red↪ NULL)) != -1) {
    switch (opt) {
        case 'y':
            options.auto_approve = true;
            break;
        case 'n':
            options.no_overwrite = true;
            break;
        case 'e':
            options.file_extension = optarg;
            break;
        case 't':
            options.thread_count = atoi(optarg);
            if (options.thread_count < 1) {
                printf("              :␣                         ␣
                   red↪                   ␣              ␣          ␣
                   red↪                                  ␣               \n
                   red↪ ");
                exit(1);
            }
            break;
        case 'h':
            print_help();
            exit(0);
        case 0:
            options.no_internet = true;
            break;
        default:
            print_help();
            exit(1);
    }
}

if (optind + 2 >= argc) {
    printf("              :␣                         ␣
       red↪                         \n");
    print_help();
    exit(1);
}

options.input_path = argv[optind];
options.dict_path = argv[optind + 1];
options.output_path = argv[optind + 2];

if (options.auto_approve && options.no_overwrite) {
    printf("              :␣              ␣                            ␣
            red↪                          ␣-y␣   ␣-n\n");
        exit(1);
    }

    return options;
}
===== FILE: ./utils/args_processing.h =====
#include "translate.h"

ProgramOptions parse_arguments(int argc, char *argv[]);
void print_help();
===== FILE: ./utils/file_processing.c =====
#include "file_processing.h"

void *translate_file_thread(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    process_file_translation(data->input_file, data->output_file,
       red↪ data->dictionary, data->options);
    free(data->input_file);
    free(data->output_file);
    free(data);
    return NULL;
}

void process_file_translation(const char *input_file, const char *
   red↪ output_file, HashTable *dictionary, ProgramOptions *options)
   red↪ {
    struct stat st;
    if (stat(output_file, &st) == 0) {
        if (options->no_overwrite) {
            printf("        ␣%s␣      ␣                        ,␣
               red↪                           ␣(            ␣-n)\n",
               red↪ output_file);
            return;
        }
        if (!options->auto_approve) {
            printf("        ␣%s␣      ␣                        .␣
               red↪                                 ?␣(y/n):␣", output_file);
            char response;
            scanf("␣%c", &response);
            if (response != 'y' && response != 'Y') {
                return;
            }
        }
    }

    FILE *in = fopen(input_file, "r");
    if (!in) {
        printf("              :␣     ␣                    ␣                   ␣
          red↪                    ␣          ␣%s\n", input_file);
        return;
    }

    FILE *out = fopen(output_file, "w");
    if (!out) {
        printf("             :␣    ␣               ␣               ␣
          red↪                    ␣          ␣%s\n", output_file);
        fclose(in);
        return;
    }

    char *line = NULL;
    size_t len = 0;
    while (getline(&line, &len, in) != −1) {
        char *translated = translate_text(line, dictionary, options,
          red↪ options−>dict_path);
        fprintf(out, "%s", translated);
        free(translated);
    }

    free(line);
    fclose(in);
    fclose(out);
}

void process_directory(const char *input_dir, const char *output_dir,
  red↪ HashTable *dictionary, ProgramOptions *options) {
    DIR *dir = opendir(input_dir);
    if (!dir) {
        printf("             :␣    ␣               ␣               ␣
          red↪                         ␣%s\n", input_dir);
        return;
    }

    struct stat st = {0};
    if (stat(output_dir, &st) == −1) {
        mkdir(output_dir, 0700);
    }

    pthread_t *threads = malloc(options−>thread_count * sizeof(
      red↪ pthread_t));
    ThreadData **thread_data = malloc(options−>thread_count * sizeof(
      red↪ ThreadData*));
    int current_thread = 0;

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) { //
      red↪
```

```
        if (entry->d_type != DT_REG) continue;

        char *ext = strrchr(entry->d_name, '.');
        if (!ext || strcmp(ext + 1, options->file_extension) != 0)
           red↪ continue;

        char input_path[PATH_MAX]; // PATH_MAX = 4096
        char output_path[PATH_MAX];
        snprintf(input_path, PATH_MAX, "%s/%s", input_dir, entry->
           red↪ d_name);
        snprintf(output_path, PATH_MAX, "%s/%s", output_dir, entry->
           red↪ d_name);

        ThreadData *data = malloc(sizeof(ThreadData));
        data->input_file = strdup(input_path);
        data->output_file = strdup(output_path);
        data->dictionary = dictionary;
        data->options = options;

        thread_data[current_thread] = data;

        if (pthread_create(&threads[current_thread], NULL,
           red↪ translate_file_thread, data) != 0) {
            printf("            :␣     ␣              ␣              ␣
               red↪           ␣       ␣           ␣%s\n", input_path);
            free(data->input_file);
            free(data->output_file);
            free(data);
            continue;
        }

        current_thread++;
        if (current_thread >= options->thread_count) { //
           red↪          >=
            for (int i = 0; i < current_thread; i++) {
                pthread_join(threads[i], NULL);
            }
            current_thread = 0;
        }
    }

    for (int i = 0; i < current_thread; i++) {
        pthread_join(threads[i], NULL);
    }

    free(threads);
    free(thread_data);
    closedir(dir);
}
```

```
===== FILE: ./utils/file_processing.h =====
#include "translate.h"

void process_file_translation(const char *input_file, const char *
   red↪ output_file, HashTable *dictionary, ProgramOptions *options);
void process_directory(const char *input_dir, const char *output_dir,
   red↪  HashTable *dictionary, ProgramOptions *options);
===== FILE: ./utils/hash_table.c =====
#include "hash_table.h"

// https://gist.github.com/MohamedTaha98/
   red↪ ccdf734f13299efb73ff0b12f7ce429f
// Djb2 hash function
unsigned long hash_function(const char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return hash;
}

HashTable *create_hash_table(size_t size) {
    HashTable *table = malloc(sizeof(HashTable));
    table->nodes = calloc(size, sizeof(HashNode*));
    table->size = size;
    return table;
}

void free_hash_table(HashTable *table) {
    for (size_t i = 0; i < table->size; i++) {
        HashNode *node = table->nodes[i];
        while (node != NULL) {
            HashNode *temp = node;
            node = node->next;
            free(temp->key);
            free(temp->value);
            free(temp);
        }
    }
    free(table->nodes);
    free(table);
}

void hash_table_insert(HashTable *table, const char *key, const char
   red↪ *value) {
    unsigned long index = hash_function(key) % table->size;
    HashNode *node = table->nodes[index];

    while (node != NULL) {
        if (strcmp(node->key, key) == 0) {
            free(node->value);
            node->value = strdup(value);
            return;
        }
        node = node->next;
    }

    HashNode *new_node = malloc(sizeof(HashNode));
    new_node->key = strdup(key);
    new_node->value = strdup(value);
    new_node->next = table->nodes[index];
    table->nodes[index] = new_node;
}

char *hash_table_search(HashTable *table, const char *key) {
    unsigned long index = hash_function(key) % table->size;
    HashNode *node = table->nodes[index];

    while (node != NULL) {
        if (strcmp(node->key, key) == 0)
            return node->value;

        node = node->next;
    }

    return NULL;
}
===== FILE: ./utils/hash_table.h =====
#include "translate.h"

unsigned long hash_function(const char *str);
HashTable *create_hash_table(size_t size);
void free_hash_table(HashTable *table);
void hash_table_insert(HashTable *table, const char *key, const char
   red↪ *value);
char *hash_table_search(HashTable *table, const char *key);
===== FILE: ./utils/internet_search.c =====
#include "internet_search.h"

struct MemoryStruct {char *memory; size_t size;};
static size_t WriteMemoryCallback(void *contents, size_t size, size_t
   red↪  nmemb, void *userp) {
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;
    char *ptr = realloc(mem->memory, mem->size + realsize + 1);
    mem->memory = ptr;
    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;
    return realsize;
}

char *search_translation_online(const char *word, const char *
   red↪ source_lang, const char *target_lang) {
    CURL *curl;
    CURLcode res;
    struct MemoryStruct chunk;

    chunk.memory = malloc(1);
    chunk.size = 0;

    curl = curl_easy_init();
    if(!curl) {
        printf("                  :␣      ␣                  ␣
           red↪                                          ␣CURL\n");
        free(chunk.memory);
        return NULL;
    }

    char url[256];
    snprintf(url, sizeof(url), "https://api.mymemory.translated.net/
       red↪ get?q=%s&langpair=%s|%s", word, source_lang, target_lang)
       red↪ ;

    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteMemoryCallback
       red↪ );
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)&chunk);
    curl_easy_setopt(curl, CURLOPT_USERAGENT, "libcurl-agent/1.0");

    res = curl_easy_perform(curl);
    if(res != CURLE_OK) {
        printf("                ␣CURL:␣%s\n", curl_easy_strerror(res));
        curl_easy_cleanup(curl);
        free(chunk.memory);
        return NULL;
    }

    curl_easy_cleanup(curl);

    json_error_t error;
    json_t *root = json_loads(chunk.memory, 0, &error);
    free(chunk.memory);

    if(!root) {
        printf("                ␣JSON:␣%s\n", error.text);
        return NULL;
    }

    //                    json'
       red↪                      ["responseData"]["translatedText"]

    json_t *responseData = json_object_get(root, "responseData");
    if(!responseData) {
        printf("            :␣      ␣responseData␣ ␣             \n")
           red↪ ;
        json_decref(root);
        return NULL;
    }

    json_t *translatedText = json_object_get(responseData, "
       red↪ translatedText");
    if(!json_is_string(translatedText)) {
        printf("            :␣      ␣                ␣ ␣
           red↪ \n");
        json_decref(root);
        return NULL;
    }

    const char *translation = json_string_value(translatedText);
    char *result = strdup(translation);

    json_decref(root);
    return result;
}

void add_word_to_dictionary(const char *word, const char *translation
   red↪ , HashTable *dictionary, const char *dict_path) {
    if (word == NULL || translation == NULL || dict_path == NULL)
       red↪ return;

    FILE *dict_file = fopen(dict_path, "a");
    if (!dict_file) {
        printf("            :␣    ␣              ␣              ␣
           red↪         ␣                ␣       ␣                   ␣
           red↪            \n");
        return;
    }

    fprintf(dict_file, "%s␣—␣%s\n", word, translation);
    fclose(dict_file);

    hash_table_insert(dictionary, word, translation);
}
===== FILE: ./utils/internet_search.h =====
#include "translate.h"
```

```
char *search_translation_online(const char *word, const char *
   red↪ source_lang, const char *target_lang);
void add_word_to_dictionary(const char *word, const char *translation
   red↪ , HashTable *dictionary, const char *dict_path);
===== FILE: ./utils/translation.c =====
#include "translation.h"

char *strdup_lower(const char *s) {
    if (!s) return NULL;
    char *result = strdup(s);
    for (int i = 0; result[i]; i++) {
        result[i] = tolower(result[i]);
    }
    return result;
}

bool is_letter(char c) { //
   red↪                                         .
    return !(c == '.' || c == ',' || c == ';' || c == ':' || c == '!'
       red↪ || c == '?' ||
             c == '"' || c == '\'' || c == '`' || c == '~' || c == '@
                red↪ ' || c == '#' ||
             c == '$' || c == '%' || c == '^' || c == '&' || c == '*'
                red↪ || c == '(' ||
             c == ')' || c == '-' || c == '_' || c == '+' || c == '='
                red↪ || c == '{' ||
             c == '}' || c == '[' || c == ']' || c == '|' || c == '\\
                red↪ ' || c == '/' ||
             c == '<' || c == '>' || c == '␣' || c == '\t' || c == '\
                red↪ n' || c == '\r');
}

bool islower_rus(int c) {
    switch (c) {
        case -48: case -103: case -90: case -93: case -102: case
           red↪ -107: case -99: case -109:
        case -88: case -87: case -105: case -91: case -86: case -83:
           red↪ case -106: case -108:
        case -101: case -98: case -96: case -97: case -112: case
           red↪ -110: case -85: case -92:
        case -81: case -89: case -95: case -100: case -104: case -94:
           red↪  case -84: case -111:
        case -82:
            return false;
        case -71: case -122: case -125: case -70: case -75: case -67:
           red↪  case -77: case -120:
        case -119: case -73: case -123: case -118: case -124: case
           red↪ -117: case -78: case -80:
```

```
        case -65: case -128: case -66: case -69: case -76: case -74:
           red↪ case -115: case -113:
        case -121: case -127: case -68: case -72: case -126: case
           red↪ -116: case -79: case -114:
            return true;

        default:
            return islower(c);
    }
}

char *apply_case(const char *word, const char *translation) {
    if (!word || !translation) return strdup(word ? word : "");

    bool all_upper = true;
    bool first_upper = isupper(word[0]);

    for (int i = 0; word[i]; i++) {
        if (islower_rus(word[i])) {
            all_upper = false;
            break;
        }
    }

    char *result = strdup(translation);
    if (all_upper) {
        for (int i = 0; result[i]; i++) {
            result[i] = toupper(result[i]);
        }
    }
    else if (first_upper && result[0]) {
        result[0] = toupper(result[0]);
        for (int i = 1; result[i]; i++) {
            result[i] = tolower(result[i]);
        }
    } else {
        for (int i = 0; result[i]; i++) {
            result[i] = tolower(result[i]);
        }
    }

    return result;
}

char *translate_word(const char *word, HashTable *dictionary,
   red↪ ProgramOptions *options, const char *dict_path) {
    if (!word || strlen(word) == 0) return strdup("");

    char *lower_word = strdup_lower(word);
    char *translation = hash_table_search(dictionary, lower_word);

    if (!translation && options && !options->no_internet) {
        if (options->auto_approve) {
            char *online_translation = search_translation_online(
               red↪ lower_word, "ru", "en");
            if (online_translation) {
                char *proper_case = apply_case(word,
                   red↪ online_translation);
                add_word_to_dictionary(lower_word, online_translation
                   red↪ , dictionary, dict_path);
                free(online_translation);
                free(lower_word);
                return proper_case;
            }
        } else if (!options->no_overwrite) {
            printf("             ␣'%s'␣     ␣                   ␣   ␣
               red↪                   .␣                 ␣   ␣
               red↪                        ?␣(y/n):␣", word);
            char response;
            scanf("␣%c", &response);
            if (response == 'y' || response == 'Y') {
                char *online_translation = search_translation_online(
                   red↪ lower_word, "ru", "en");
                if (online_translation) {
                    char *proper_case = apply_case(word,
                       red↪ online_translation);
                    add_word_to_dictionary(lower_word,
                       red↪ online_translation, dictionary, dict_path
                       red↪ );
                    free(online_translation);
                    free(lower_word);
                    return proper_case;
                }
            }
        }
    }

    char *result = translation ? apply_case(word, translation) :
       red↪ strdup(word);
    free(lower_word);
    return result;
}

char *translate_text(const char *text, HashTable *dictionary,
   red↪ ProgramOptions *options, const char *dict_path) {
    if (!text) return strdup("");

    size_t text_len = strlen(text);
    char *result = calloc(text_len * 3 + 1, 1);
    size_t result_pos = 0;

    size_t word_start = 0;
    bool in_word = false;

    for (size_t i = 0; i <= text_len; i++) {
        char c = text[i];
        bool is_let = is_letter(c);

        if (!in_word && is_let) {
            word_start = i;
            in_word = true;
        } else if (in_word && (!is_let || c == '\0')) {
            size_t word_len = i - word_start;
            char *word = malloc(word_len + 1);
            strncpy(word, text + word_start, word_len);
            word[word_len] = '\0';

            char *translated = translate_word(word, dictionary,
               red↪ options, dict_path);
            size_t trans_len = strlen(translated);
            strncpy(result + result_pos, translated, trans_len);
            result_pos += trans_len;

            free(translated);
            free(word);
            in_word = false;
        }

        if (!is_let && c != '\0') {
            result[result_pos++] = c;
        }
    }

    result[result_pos] = '\0';
    return result;
}
===== FILE: ./utils/translation.h =====
#include "translate.h"

char *translate_word(const char *word, HashTable *dictionary,
   red↪ ProgramOptions *options, const char *dict_path);
char *translate_text(const char *text, HashTable *dictionary,
   red↪ ProgramOptions *options, const char *dict_path);
```