Харківський університет радіоелектроніки
Факультет комп'ютерних наук
Кафедра програмної інженерії

ЗВІТ

до практичного заняття №1 з дисципліни
"Безпека програм та даних"
на тему "Шифр Цезаря"

Виконав ст. гр ПЗПІ-20-2
Овчаренко Михайло Миколайович

Перевірив
асистент кафедри ПІ
Олійник О. О.

Харків 2023

МЕТА

Ознайомити студентів з шифром Цезаря та Віженера, відпрацювати навички використовування цих шифрів для кодування та декодування тексту.

ЗАВДАННЯ

Розробити програму для шифрування та дешифрування Unicode тексту в кодовій сторінці UTF-8 методами алгоритмів Цезаря та Віженера. Імплементувати відповідні алгоритми Цезаря та Віженера.

ХІД РОБОТИ

Нехай вхідні дані програми складаються з двох файлів: текста для шифрації або дешифрації, та текст ключа. Обидва тексти представленно в кодовій сторінці UTF-8. А оскільки алгоритми працюватимуть саме із буквами з тексту, то мною було створено функції для обробки байтових послідовностей UTF-8 (див. додаток А, файл “utf.c”). Функція “utf8_codepoint_length” знаходить довжину наступної UTF-8 послідовності, перевіряючи старші біти поточного байту. Функція “utf8_decode” приймає послідовність байтів та конвертує їх у кодову точку стандарту Unicode, універсального стандарту для усіх кодових сторінок. “utf8_encode” робить те ж саме, тільки навпаки. Знаючи кодову точку букви ми можемо зручно робити арифметичні дії та зсуви в рамках алфавітів, оскільки алфавіт зазвичай розміщено цілком послідовно. Дуже зручно це робити, наприклад, з російським алфавітом, з текстом котрого програма і буде працювати (дивитись

розділ Unicode для кирилічних символів з U+0400 по U+04FF, https://www.utf8-chartable.de/unicode-utf8-table.pl).

Визначимо макроси для роботи з російським алфавітом в межах Unicode:

```
1  #define ALPHABET_LENGTH 31
2  #define FIRST_CAPITAL_LETTER_HEX 0x410
3  #define FIRST_LOWERCASE_LETTER_HEX 0x430
```

Тепер можемо приступити до створення процедур шифрування та дешифрування для кожного з алгоритмів. Повний їх код можна побачити в додатку А, файл “main.c”.

Перейдемо до тестування коректності роботи процедур алгоритма Цезаря. Файл “test.h”, що реалізує тестування усіх функцій, наведено в додатку А. Частково зміст оригінального текстового файлу видно на рис. 1. Результат зсуву на 18 літер можна побачити на рис. 2. Розшифрувавши файл, зсунувши на 18 літер назад, отримуємо те, що зображено на рис. 3.

```
Я сначала подумала, что он – секретный физик. Такое же загадочное выражение лица,
встрепанные волосы и небрежность в одежде. Представляете, вообще не посмотрел в мою сторону!
Меня это так возмутило. Сначала я в пику ему стала вовсю флиртовать с поклонниками, ходила
танцевать, один раз даже сбила юбкой бутылку пива ему на колени. Так он и в этом случае на
меня не посмотрел. Поднял глаза на хозяйку квартиры и говорит: "Лена, мне бы произвести
процесс дегидратации штанов". Лена долго пыталась понять – что она должна сделать с его
штанами, но потом сообразила, увела Сережу в ванную, откуда он вернулся в штанах ее
мужа-культуриста. И черт меня дернул выйти за него замуж! Ведь сама - не дура! Не уродина! И
```

Рисунок 1 – Оригінальний текст

```
С гятйтэт бацеютэт, йда ая - гчьвчдяны жъщъь. Дтьач шч щтхтцайяач фнвтшчяьч эъит,
фгдвчбтяянч фаэагн ъ ячувчшяагдо ф ацчшцч. Бвчцгдтфэсчдч, фааулч яч багюадвчэ ф юар гдаваяе!
Ючяс пда дть фащюедъэа. Гятйтэт с ф бъье чюе гдтэт фафгр жэъвдафтдо г баьэаяяььтюъ, зацъэт
дтяичфтдо, ацъя втщ цтшч гуъэт руьаы уеднэье бъфт чюе ят ьаэчяъ. Дть ая ъ ф пдаю гэейтч ят
ючяс яч багюадвчэ. Бацясэ хэтщт ят защсыье ьфтвдъвн ъ хафавъд: "Эчят, юяч ун бваъщфчгдъ
бваичгг цчхъцвтдтиъь кдтяаф". Эчят цаэха бндтэтго баясдо - йда аят цаэшят гцчэтдо г чха
кдтятюъ, яа бадаю гааувтщъэт, ефчэт Гчвчше ф фтяяер, адьецт ая фчвяеэгс ф кдтятз чч
юешт-ьеэодевъгдт. Ъ йчвд ючяс цчвяеэ фныдъ щт ячха щтюеш! Фчцо гтют - яч цевт! Яч евацъят! Ъ
```

Рисунок 2 – Шифрований текст

```
Я сначала подумала, что он - секретный физик. Такое же загадочное выражение лица,
встрепанные волосы и небрежность в одежде. Представляете, вообще не посмотрел в мою сторону!
Меня это так возмутило. Сначала я в пику ему стала вовсю флиртовать с поклонниками, ходила
танцевать, один раз даже сбила юбкой бутылку пива ему на колени. Так он и в этом случае на
меня не посмотрел. Поднял глаза на хозяйку квартиры и говорит: "Лена, мне бы произвести
процесс дегидратации штанов". Лена долго пыталась понять - что она должна сделать с его
штанами, но потом сообразила, увела Сережу в ванную, откуда он вернулся в штанах ее
мужа-культуриста. И черт меня дернул выйти за него замуж! Ведь сама - не дура! Не уродина! И
```

Рисунок 3 – Розшифроване повідомлення, що цілком співпадає з початковим

З метою наочності та простоти, перевіримо, чи є різниця між початковим та дешифрованим файлом за допомогою Unix-утиліти diff (див. рис. 4). Якщо символи файлів не співпадають, програма обов’яково повідомить нам про це.

```
User@DESKTOP-ERVQ9EV /cygdrive/d/Univer/4th_Course_1_sem/CryptoIS/PZ1/test
$ diff --normal test1-original.txt test1-decrypted.txt

User@DESKTOP-ERVQ9EV /cygdrive/d/Univer/4th_Course_1_sem/CryptoIS/PZ1/test
$
```

Рисунок 4 – Перевірка різниці між файлами – різниці немає

Бачимо, що програма нічого не відповіла, а отже, два файли є повністю ідентичні, тобто початковий текст було успішно дешифровано повністю.

Перейдемо до тестування шифром Віженера. Покладемо у файл “vigenere-test1.txt” такий текст: “*Первым об упрощении этой схемы задумался Ади Шамир в 1984 году. Он предположил, что если бы появилась возможность использовать в качестве открытого ключа имя или почтовый адрес Алисы, то это лишило бы сложную процедуру аутентификации всякого смысла. Долгое время идея Шамира оставалась всего лишь красивой криптографической головоломкой, но в 2000 году, благодаря одной известной уязвимости в эллиптической криптографии, идею удалось воплотить в жизнь.*”.

Створимо файл ключа “vigenere-key.txt” з таким вмістом: “*алиса*”. Після запуску програми бачимо, що текст шифру є таким: “*Пршуым щй дпрщбциу егой ьэцмы тихумлувя Апр Йамуш у 1984 гопы. Ян пынхпоццчил, въя есцр ты пщзуиллщн вотфяжнщщгь иьчяльтцуатз к ыачрщгве щъырыэцфо кцжиа ичз щли ъцитонгъ адынв Алущм, то иъя лигрьо бж щьожшып прщюцдуыы сутрхгифутсциу квякщля смжщьа. Дщуфое ншцмя умця Шлфщра щщгавлуссь ншцго црйь кыививщс ыриъъягрльщчеьтяй гщуявоццэкоф, хя в 2000 гщмд, бллядаыз яднщс щзврщгноф ырзвуфясту к оллучгичрщыой хшщптщлбафур, щдей ыхалщщн воъуятиэд у житхн.*”. Тепер розшифруємо його тим самим ключем та зробимо перевірку за допомогою diff (див. рис. 5).

```
User@DESKTOP-ERVQ9EV /cygdrive/d/Univer/4th_Course_1_sem/CryptoIS/PZ1/test
$ cat vigenere-decrypted.txt
Первым об упрощении этой схемы задумался Ади Шамир в 1984 году. Он предположил,
 что если бы появилась возможность использовать в качестве открытого ключа имя
или почтовый адрес Алисы, то это лишило бы сложную процедуру аутентификации вся
кого смысла. Долгое время идея Шамира оставалась всего лишь красивой криптограф
ической головоломкой, но в 2000 году, благодаря одной известной уязвимости в эл
липтической криптографии, идею удалось воплотить в жизнь.
User@DESKTOP-ERVQ9EV /cygdrive/d/Univer/4th_Course_1_sem/CryptoIS/PZ1/test
$ diff --normal vigenere-decrypted.txt vigenere-test1.txt

User@DESKTOP-ERVQ9EV /cygdrive/d/Univer/4th_Course_1_sem/CryptoIS/PZ1/test
$
```

Рисунок 5 – Доказ відповідності файлів

Таким чином, усі процедури, як шифрування, так і дешифрування тексту, відпрацювали так, як потрібно.

ВИСНОВКИ

Розробив програму для шифрування та дешифрування Unicode тексту в кодовій сторінці UTF-8 методами алгоритмів Цезаря та Віженера. Імплементував відповідні алгоритми Цезаря та Віженера мовою програмування C.

ДОДАТОК А

Програмний код

Файл utf.c

```
 1  #include <stdint.h>
 2
 3  int utf8_codepoint_length(const uint8_t byte) {
 4      if ((byte & 0x80) == 0) {
 5          return 1; // Однобайтовая кодовая точка
 6      } else if ((byte & 0xE0) == 0xC0) {
 7          return 2; // Двухбайтовая кодовая точка
 8      } else if ((byte & 0xF0) == 0xE0) {
 9          return 3; // Трехбайтовая кодовая точка
10      } else if ((byte & 0xF8) == 0xF0) {
11          return 4; // Четырехбайтовая кодовая точка
12      } else {
13          return -1; // Недопустимая последовательность
14      }
15  }
16
17  void utf8_encode(uint32_t codepoint, uint8_t utf8_sequence[4], int* length)
{
18      if (codepoint <= 0x7F) {
19          // Single-byte character
20          utf8_sequence[0] = (uint8_t)codepoint;
21          *length = 1;
22      } else if (codepoint <= 0x7FF) {
23          // Two-byte character
24          utf8_sequence[0] = (uint8_t)((codepoint >> 6) | 0xC0);
25          utf8_sequence[1] = (uint8_t)((codepoint & 0x3F) | 0x80);
26          *length = 2;
27      } else if (codepoint <= 0xFFFF) {
28          // Three-byte character
```

```
        utf8_sequence[0] = (uint8_t)((codepoint >> 12) | 0xE0);
        utf8_sequence[1] = (uint8_t)(((codepoint >> 6) & 0x3F) | 0x80);
        utf8_sequence[2] = (uint8_t)((codepoint & 0x3F) | 0x80);
        *length = 3;
    } else if (codepoint <= 0x10FFFF) {
        // Four-byte character
        utf8_sequence[0] = (uint8_t)((codepoint >> 18) | 0xF0);
        utf8_sequence[1] = (uint8_t)(((codepoint >> 12) & 0x3F) | 0x80);
        utf8_sequence[2] = (uint8_t)(((codepoint >> 6) & 0x3F) | 0x80);
        utf8_sequence[3] = (uint8_t)((codepoint & 0x3F) | 0x80);
        *length = 4;
    } else {
        // Invalid code point
        *length = 0;
    }
}

uint32_t utf8_decode(const uint8_t* sequence, int length) {
    if (length <= 0) {
        return 0;
    }
    uint32_t codepoint = 0;
    if (length == 1) {
        // Однобайтовая кодовая точка
        codepoint = sequence[0];
    } else {
        // Многобайтовая кодовая точка
        codepoint = sequence[0] & (0xFF >> (length + 1));
        for (int i = 1; i < length; i++) {
            codepoint <<= 6;
            codepoint |= (sequence[i] & 0x3F);
        }
    }
    return codepoint;
}
```

Файл main.c

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

#define CYRILLIC_CODE_POINT_START_HEX 0x400
#define CYRILLIC_CODE_POINT_END_HEX 0x4FF
#define ALPHABET_LENGTH 31
#define FIRST_CAPITAL_LETTER_HEX 0x410
#define FIRST_LOWERCASE_LETTER_HEX 0x430
#define FIRST_CAPITAL_LETTER_LOW_BYTE_HEX 0x90
#define FIRST_LOWERCASE_LETTER_LOW_BYTE_HEX 0xb0

extern int utf8_codepoint_length(const uint8_t byte);
extern uint32_t utf8_decode(const uint8_t* sequence, int length);
extern void utf8_encode(uint32_t codepoint, uint8_t utf8_sequence[4], int*
length);

long get_file_size(FILE* fp){
        fseek(fp, 0, SEEK_END);
    long file_size = ftell(fp);
    fseek(fp, 0, SEEK_SET);
    return file_size;
}

int inside_capitals_ru(uint32_t point){
        return point >= FIRST_CAPITAL_LETTER_HEX &&
                                  point <= (FIRST_CAPITAL_LETTER_HEX +
ALPHABET_LENGTH);
}

int inside_lowercase_ru(uint32_t point){
        return point >= FIRST_LOWERCASE_LETTER_HEX &&
```

```
    33                                      point <= (FIRST_LOWERCASE_LETTER_HEX +
ALPHABET_LENGTH);
    34  }
    35
    36  // ceasar_encrypt
    37  // IN: текст російською абеткою в UTF-8, цифровий ключ
    38  // OUT: зашифрований текст
    39  char* caesar_encrypt(char* textUTF8, long buf_size, int key){
    40          // такий самий за розміром буфер
    41          char* encrypted = (char*)malloc(buf_size);
    42          // Ітеруємось по усіх utf-8 послідовностях байтів, перевіряємо чи
являє собою букву російської абетки.
    43          // Якщо це не так, тоді пропускаємо як спец. символ та вважаємо
його з блоку Basic Latin
    44          // Інакше, виконуємо шифрування алгоритмом Цезаря та пишемо в буфер
    45          for(long i = 0; i < buf_size; ){
    46                  int codepoint_length = utf8_codepoint_length(textUTF8[i]);
    47                  // Якщо односимвольна послідовність, записуємо спец. символ
та йдемо далі
    48                  if (codepoint_length == 1 || codepoint_length == -1) {
    49                          encrypted[i] = textUTF8[i];
    50                          i++;
    51                          continue;
    52                  }
    53                  // рахуємо кодову точку
    54                  uint32_t point = utf8_decode(textUTF8 + i,
codepoint_length);
    55                  // робимо обчислення спираючись саме на кодову точку,
    56                  // а не на байтове представлення
    57                  unsigned int encr = (point + key);
    58                  unsigned int offset = 0;
    59                  // нехай codepoint_legth зараз є 2
    60                  if (codepoint_length == 2 && inside_capitals_ru(point)) {
    61                          // саме шифрування молодшого байту робить зрушення
вперед за абеткою
    62                          // модулем буде довжина абетки (31 буква) + номер
початкової букви абетки (А)
```

```
    63                          if (encr >= ALPHABET_LENGTH +
FIRST_CAPITAL_LETTER_HEX + 1){
    64                                  offset = FIRST_CAPITAL_LETTER_HEX;
    65                          }
    66                          encr = encr % (ALPHABET_LENGTH +
FIRST_CAPITAL_LETTER_HEX + 1);
    67                          encr += offset;
    68                          int res_codepoint_len = 0;
    69                          uint8_t res_codepoint_bytes[4];
    70                          memset(res_codepoint_bytes, 0, 4);
    71                          utf8_encode(encr, res_codepoint_bytes,
&res_codepoint_len);
    72                          for(int j = 0; j < res_codepoint_len; j++){
    73                                  encrypted[i + j] = res_codepoint_bytes[j];
    74                          }
    75                          i += res_codepoint_len;
    76                          continue;
    77                  }
    78                  else if(codepoint_length == 2 &&
inside_lowercase_ru(point)) {
    79                          if (encr >= ALPHABET_LENGTH +
FIRST_LOWERCASE_LETTER_HEX + 1){
    80                                  offset = FIRST_LOWERCASE_LETTER_HEX;
    81                          }
    82                          encr = encr % (ALPHABET_LENGTH +
FIRST_LOWERCASE_LETTER_HEX + 1);
    83                          encr += offset;
    84                          int res_codepoint_len = 0;
    85                          uint8_t res_codepoint_bytes[4];
    86                          memset(res_codepoint_bytes, 0, 4);
    87                          utf8_encode(encr, res_codepoint_bytes,
&res_codepoint_len);
    88                          for(int j = 0; j < res_codepoint_len; j++){
    89                                  encrypted[i + j] = res_codepoint_bytes[j];
    90                          }
    91                          i += res_codepoint_len;
    92                          continue;
    93                  }
```

```
    94                  else{ // UTF-послідовність явно не являє собою Basic Latin,
але ми просто його залишаємо
    95                          for(int j = 0; j < codepoint_length; j++)
    96                                  encrypted[i + j] = textUTF8[i + j];
    97                          i += codepoint_length;
    98                          continue;
    99                  }
   100          }
   101          return encrypted;
   102  }
   103
   104  // caesar_decrypt
   105  // IN: шифр, ключ
   106  // OUT: текст російською абеткою в UTF-8
   107  char* caesar_decrypt(char* cipheredText, long buf_size, int key){
   108          char* decrypted = (char*) malloc(buf_size);
   109          for(int i = 0; i < buf_size; ){
   110                  int codepoint_len = utf8_codepoint_length(cipheredText[i]);
   111                  if(codepoint_len == 1 || codepoint_len == -1){
   112                          decrypted[i] = cipheredText[i];
   113                          i++;
   114                          continue;
   115                  }
   116                  uint32_t point = utf8_decode(cipheredText + i,
codepoint_len);
   117                  uint32_t decr = point - key;
   118                  uint32_t result = 0;
   119                  uint32_t offset = 0;
   120                  if(codepoint_len == 2 && // буква належить до нижнього
регістру
   121                          inside_lowercase_ru(point)){
   122                          if(decr < FIRST_LOWERCASE_LETTER_HEX)
   123                                  offset = (FIRST_LOWERCASE_LETTER_HEX -
decr) % ALPHABET_LENGTH;
   124                          if(offset > 0)
   125                                  result = FIRST_LOWERCASE_LETTER_HEX +
ALPHABET_LENGTH + 1 - offset;
```

```
126                             else result = decr;
127                             int point_len = 0;
128                             uint8_t seq[4];
129                             memset(seq, 0, 4);
130                             utf8_encode(result, seq, &point_len);
131                             for(int j = 0; j < point_len; j++){
132                                     decrypted[i + j] = seq[j];
133                             }
134                             i += point_len;
135                             continue;
136                     }
137                     else if(codepoint_len == 2 && inside_capitals_ru(point)){
138                             if(decr < FIRST_CAPITAL_LETTER_HEX)
139                                     offset = (FIRST_CAPITAL_LETTER_HEX - decr)
% ALPHABET_LENGTH;
140                             if(offset > 0)
141                                     result = FIRST_CAPITAL_LETTER_HEX +
ALPHABET_LENGTH + 1 - offset;
142                             else result = decr;
143                             int point_len = 0;
144                             uint8_t seq[4];
145                             memset(seq, 0, 4);
146                             utf8_encode(result, seq, &point_len);
147                             for(int j = 0; j < point_len; j++){
148                                     decrypted[i + j] = seq[j];
149                             }
150                             i += point_len;
151                             continue;
152                     }
153                     else{ // UTF-послідовність явно не являє собою Basic Latin,
але ми просто його залишаємо
154                             for(int j = 0; j < codepoint_len; j++)
155                                     decrypted[i + j] = cipheredText[i + j];
156                             i += codepoint_len;
157                             continue;
158                     }
```

```
159          }
160          return decrypted;
161  }
162
163  // vigenere_encrypt
164  // IN: string message, string keyphrase
165  // OUT: encrypted message string
166  char* vigenere_encrypt(const char* msg, uint32_t msglen, char* key,
uint32_t keylen){
167          char* encrbuf = (char*) malloc(msglen);
168          uint32_t k = 0;
169          for(uint32_t i = 0; i < msglen; ){
170                  int seqlen = utf8_codepoint_length(msg[i]);
171                  if(seqlen == 1){
172                          encrbuf[i] = msg[i];
173                          i++;
174                          continue;
175                  }
176                  uint32_t point = utf8_decode(msg + i, seqlen);
177                  uint32_t key_index = k % keylen;
178                  uint32_t keyseqlen = utf8_codepoint_length(key[key_index]);
179                  if (keyseqlen < 2) {
180                          fprintf(stderr, "key UTF-8 sequence is less than 2
at %d.\n", key_index);
181                          // Do nothing
182                  }
183                  uint32_t point_key = utf8_decode(key + key_index,
keyseqlen);
184                  k += keyseqlen;
185                  uint32_t offset_key = 0;
186                  if(inside_capitals_ru(point_key)){
187                          offset_key = point_key - FIRST_CAPITAL_LETTER_HEX;
188                  }
189                  else if(inside_lowercase_ru(point_key)){
190                          offset_key = point_key -
FIRST_LOWERCASE_LETTER_HEX;
191                  }
```

```
                uint32_t offset = 0;
                uint32_t encoded = point + offset_key;
                if(seqlen == 2 && inside_capitals_ru(point)){
                        if(encoded > FIRST_CAPITAL_LETTER_HEX +
ALPHABET_LENGTH){
                                offset += FIRST_CAPITAL_LETTER_HEX;
                        }
                        encoded = encoded % (FIRST_CAPITAL_LETTER_HEX +
ALPHABET_LENGTH + 1);
                        encoded += offset;
                        int len = 0;
                        uint8_t sequtf[4];
                        memset(sequtf, 0, 4);
                        utf8_encode(encoded, sequtf, &len);
                        for(int j = 0; j < len; j++){
                                encrbuf[i + j] = sequtf[j];
                        }
                        i += len;
                        continue;
                }
                else if(seqlen == 2 && inside_lowercase_ru(point)){
                        if(encoded > FIRST_LOWERCASE_LETTER_HEX +
ALPHABET_LENGTH){
                                offset += FIRST_LOWERCASE_LETTER_HEX;
                        }
                        encoded = encoded % (FIRST_LOWERCASE_LETTER_HEX +
ALPHABET_LENGTH + 1);
                        encoded += offset;
                        int len = 0;
                        uint8_t sequtf[4];
                        memset(sequtf, 0, 4);
                        utf8_encode(encoded, sequtf, &len);
                        for(int j = 0; j < len; j++){
                                encrbuf[i + j] = sequtf[j];
                        }
                        i += len;
                        continue;
```

```
225                 }
226                 else{
227                         for(int j = 0; j < seqlen; j++){
228                                 encrbuf[i + j] = msg[i + j];
229                         }
230                         i += seqlen;
231                         continue;
232                 }
233         }
234         return encrbuf;
235 }
236
237 char* vigenere_decrypt(const char* cyph, uint32_t cyphlen, char* key,
uint32_t keylen){
238         char* msg = (char*) malloc(cyphlen);
239         uint32_t k = 0;
240         for(int i = 0; i < cyphlen; ){
241                 int cplen = utf8_codepoint_length(cyph[i]);
242                 if(cplen == 1){
243                         msg[i] = cyph[i];
244                         i++;
245                         continue;
246                 }
247                 uint32_t cp = utf8_decode(cyph + i, cplen);
248                 uint32_t keyind = k % keylen;
249                 int cpkeylen = utf8_codepoint_length(key[keyind]);
250                 uint32_t cpkey = utf8_decode(key + (keyind), cpkeylen);
251                 k += cpkeylen;
252                 uint32_t keyoffset = 0;
253                 if(inside_capitals_ru(cpkey))
254                         keyoffset = cpkey - FIRST_CAPITAL_LETTER_HEX;
255                 else if(inside_lowercase_ru(cpkey))
256                         keyoffset = cpkey - FIRST_LOWERCASE_LETTER_HEX;
257                 uint32_t diff = cp - keyoffset;
258                 uint32_t offset_alphabet = 0;
```

```
259                     uint32_t resultcp = 0;
260                     if(cplen == 2 && inside_capitals_ru(cp)){
261                             if(diff < FIRST_CAPITAL_LETTER_HEX){
262                                     offset_alphabet = (FIRST_CAPITAL_LETTER_HEX
- diff) % ALPHABET_LENGTH;
263                                     resultcp = (FIRST_CAPITAL_LETTER_HEX +
ALPHABET_LENGTH + 1 - offset_alphabet);
264                             }
265                             else resultcp = diff;
266                             int rescplen = 0;
267                             uint8_t res_seq[4];
268                             utf8_encode(resultcp, res_seq, &rescplen);
269                             for(int j = 0; j < rescplen; j++){
270                                     msg[i + j] = res_seq[j];
271                             }
272                             i += rescplen;
273                             continue;
274                     }
275                     else if(cplen == 2 && inside_lowercase_ru(cp)){
276                             if(diff < FIRST_LOWERCASE_LETTER_HEX){
277                                     offset_alphabet =
(FIRST_LOWERCASE_LETTER_HEX - diff) % ALPHABET_LENGTH;
278                                     resultcp = (FIRST_LOWERCASE_LETTER_HEX +
ALPHABET_LENGTH + 1 - offset_alphabet);
279                             }
280                             else resultcp = diff;
281                             int rescplen = 0;
282                             uint8_t res_seq[4];
283                             utf8_encode(resultcp, res_seq, &rescplen);
284                             for(int j = 0; j < rescplen; j++){
285                                     msg[i + j] = res_seq[j];
286                             }
287                             i += rescplen;
288                             continue;
289                     }
290                     else{
291                             for(int j = 0; j < cplen; j++){
```

```
292                                 msg[i + j] = cyph[i + j];
293                         }
294                         i += cplen;
295                         continue;
296                 }
297         }
298         return msg;
299 }
300
301
302 #include "test.h"
303
304 int main(int argc, char const *argv[])
305 {
306         UINT oldCodePage;
307
308         oldCodePage = GetConsoleOutputCP();
309     if (!SetConsoleOutputCP(65001)) {
310         perror("Error setting CP.\n");
311         return 1;
312     }
313
314 // Тести
315     test_caesar_encrypt_decrypt();
316         test_vigenere_encrypt_decrypt();
317
318     SetConsoleOutputCP(oldCodePage);
319     return 0;
320 }
```

Код файлу test.h

```
1  int test_caesar_encrypt_rot1(){
2          FILE* fp = fopen("./test/test2.txt", "rb");
3          if(fp == NULL) {
```

```
 4                 perror("Error opening test2.txt.\n");
 5                 return 1;
 6         }
 7         long buf_size = get_file_size(fp);
 8         char* buf = (char*) malloc(buf_size);
 9         long read = fread(buf, 1, buf_size, fp);
10         if (read != buf_size){
11                 perror("Read less bytes than buf_size.\n");
12                 return 1;
13         }
14         char* crypted = caesar_encrypt(buf, buf_size, 1);
15         FILE* fp_res = fopen("./test/test2-encr.txt", "wb");
16         if (fp_res == NULL){
17                 perror("Error creating test2-encr.txt.\n");
18                 return 1;
19         }
20         fwrite(crypted, 1, buf_size, fp_res);
21         free(buf);
22         free(crypted);
23         fclose(fp);
24         fclose(fp_res);
25         return 0;
26 }
27
28 void test_caesar_encrypt_decrypt(){
29         FILE* fp = fopen("./test/test1-original.txt", "rb");
30         if (fp == NULL) {
31         perror("Error opening the file.\n");
32         return;
33     }
34
35     size_t file_size = get_file_size(fp);
36     char* utf8_original_buffer = (char*)malloc(file_size);
37     size_t read = fread(utf8_original_buffer, 1, file_size, fp);
38     if (file_size != read){
```

```
39          perror("test1-original wasn't read to the end.\n");
40          return;
41      }
42
43          char* result = caesar_encrypt(utf8_original_buffer, file_size, 18);
44      FILE* fp_result = fopen("./test/test1-encrypted.txt", "wb");
45      fwrite(result, 1, file_size, fp_result);
46
47      char* decrypted = caesar_decrypt(result, file_size, 18);
48      FILE* fp_decr = fopen("./test/test1-decrypted.txt", "wb");
49      fwrite(decrypted, 1, file_size, fp_decr);
50
51      fclose(fp_decr);
52      free(decrypted);
53      fclose(fp_result);
54      free(result);
55
56      free(utf8_original_buffer);
57      fclose(fp);
58  }
59
60  void test_vigenere_encrypt_decrypt(){
61          FILE* fp = fopen("./test/vigenere-test1.txt", "rb");
62          if(fp == NULL){
63                  perror("Error opening vigenere-test1.txt.\n");
64                  return;
65          }
66          long file_size = get_file_size(fp);
67          char* test1 = (char*) malloc(file_size);
68          long read = fread(test1, 1, file_size, fp);
69          if(read != file_size){
70                  perror("vigener-test1.txt wasn't read to the end.\n");
71                  return;
72          }
73          FILE* fp_key = fopen("./test/vigenere-key.txt", "rb");
```

```
    74          if(fp_key == NULL){
    75                  perror("Error opening vigenere-key.txt.\n");
    76                  return;
    77          }
    78          long key_file_size = get_file_size(fp_key);
    79          char* key1 = (char*) malloc(key_file_size);
    80          long key_read = fread(key1, 1, key_file_size, fp_key);
    81          if(key_read != key_file_size){
    82                  perror("vigener-key.txt wasn't read to the end.\n");
    83                  return;
    84          }
    85          char* res1 = vigenere_encrypt(test1, file_size, key1,
key_file_size);
    86          FILE* fp_out = fopen("./test/vigenere-encrypted.txt", "wb");
    87          if(fp_out == NULL){
    88                  perror("Error opening vigenere-encrypted.txt for write
binary.\n");
    89                  return;
    90          }
    91          fwrite(res1, 1, file_size, fp_out);
    92
    93          char* res0 = vigenere_decrypt(res1, file_size, key1,
key_file_size);
    94          FILE* fp_dec = fopen("./test/vigenere-decrypted.txt", "wb");
    95          if(fp_dec == NULL){
    96                  perror("Error opening vigenere-decrypted.txt for write
binary.\n");
    97                  return;
    98          }
    99          //fwrite(res0, 1, file_size, stdout);
   100          fwrite(res0, 1, file_size, fp_dec);
   101
   102          fclose(fp);
   103          fclose(fp_key);
   104          fclose(fp_out);
   105          fclose(fp_dec);
   106          free(res0);
```

```
107         free(test1);
108         free(res1);
109         free(key1);
110 }
111
112 void test_vigenere_decrypt(){
113         FILE* fp = fopen("./test/vigenere-encrypted.txt", "rb");
114         if(fp == NULL){
115                 perror("Error opening vigenere-encrypted.txt.\n");
116                 return;
117         }
118         long size = get_file_size(fp);
119         char* enc = (char*) malloc(size);
120         long read = fread(enc, 1, size, fp);
121         if(read != size){
122                 perror("vigenere-encrypted.txt wasn't read to the end");
123                 return;
124         }
125         FILE* fp_key = fopen("./test/vigenere-key.txt", "rb");
126         if(fp_key == NULL){
127                 perror("Error opening vigenere-key.txt.\n");
128                 return;
129         }
130         long key_file_size = get_file_size(fp_key);
131         char* key1 = (char*) malloc(key_file_size);
132         long key_read = fread(key1, 1, key_file_size, fp_key);
133         if(key_read != key_file_size){
134                 perror("vigener-key.txt wasn't read to the end.\n");
135                 return;
136         }
137         char* dec = vigenere_decrypt(enc, size, key1, key_file_size);
138         FILE* fp_dec = fopen("./test/vigenere-decrypted.txt", "wb");
139         if(fp_dec == NULL){
140                 perror("Error opening vigenere-decrypted.txt for write
binary.\n");
```

```
141                     return;
142             }
143             fwrite(dec, 1, size, fp_dec);
144
145             fclose(fp_dec);
146             fclose(fp_key);
147             fclose(fp);
148             free(dec);
149             free(enc);
150             free(key1);
151     }
```