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

Кафедра вычислительных систем

ОТЧЕТ
по практической работе 3

по дисциплине «**Программирование**»

Выполнил:
студент гр. ИС-241
«25» апреля 2023 г. _____ /Олейников Д.О./

Проверил:
Старший преподаватель
«25» апреля 2018 г. _____ /Фульман Д.О./

Оценка «_____»

Новосибирск 2023

ОГЛАВЛЕНИЕ

ЗАДАНИЕ

Работа состоит из двух частей.

В первой части вам предлагается разобрать алгоритм кодирования целых чисел, называемый varint (**var**iable **int**eger). Такой способ кодирования позволяет использовать переменное количество байт для представления целых чисел и благодаря этому обеспечивает компактность данных. Вам нужно разработать приложение для записи и чтения чисел в сыром виде и в формате varint, сравнить эти два способа по эффективности и сделать выводы о применимости предложенного способа кодирования.

Во второй части необходимо самостоятельно реализовать алгоритм кодирования UTF-8. Этот алгоритм решает аналогичную задачу — позволяет кодировать целые числа переменным количеством байт, но используется для кодирования кодов символов и поддерживает обратную совместимость с кодировкой ASCII.

Задание 1

Разработайте приложение, которое генерирует 1000000 случайных чисел и записывает их в два бинарных файла. В файл `uncompressed.dat` запишите числа в несжатом формате, в файл `compressed.dat` — в формате varint. Сравните размеры файлов.

Реализуйте чтение чисел из двух файлов. Добавьте проверку: последовательности чисел из двух файлов должны совпадать.

Использование формата varint наиболее эффективно в случаях, когда подавляющая доля чисел имеет небольшие значения. Для выполнения работы используйте функцию генерации случайных чисел:

Задание 2

Разработать приложение для кодирования и декодирования чисел по описанному выше алгоритму.

Пример кодирования:

```
$ ./coder encode points.txt units.bin
```

ВЫПОЛНЕНИЕ РАБОТЫ

1 Задание.

В данном задание нам требуется 3 функции: для кодирования чисел, для декодирование, и для генерации случайного числа.

```c
size_t encode_varint(uint32_t value, uint8_t *buf)
{
    assert(buf != NULL);
    uint8_t *cur = buf;
    while (value >= 0x80)
    {
        const uint8_t byte = (value & 0x7f) | 0x80;
        *cur = byte;
        value >>= 7;
        ++cur;
    }
    *cur = value;
    ++cur;
    return cur - buf;
}
```

Функция **encode_varint** кодирует целочисленное значение value в формате Base 128 Varint и сохраняет результат в буфер buf.

В самом начале идёт проверка, с помощью макроса assert, равен ли buf NULL.

Переменная cur указывает на текущую позицию в buf. В цикле while проверяется, является ли значение value больше или равным 0x80. Если это так, значит, должен записаться ещё один байт в буфер, т.к. каждый байт в формате Base 128 Varint содержит 7 бит данных и 1 бит продолжения. Создаётся byte, который содержит младшие 7 битов значения value и бит продолжения (0x80). Затем записывается этот в текущую позицию cur в буфере buf, сдвигается значение value на 7 бит вправо и увеличивается значение cur на 1. Когда значение value становится меньше, чем 0x80, записывается оставшееся значение value в текущую позицию в буфере buf и увеличивается значение cur на 1.

```c
uint32_t decode_varint(const uint8_t **bufp)
{
    const uint8_t *cur = *bufp;
    uint8_t byte = *cur++;
    uint32_t value = byte & 0x7f;
    size_t shift = 7;
    while (byte >= 0x80)
    {
        byte = *cur++;
        value += (byte & 0x7f) << shift;
        shift += 7;
    }
    *bufp = cur;
    return value;
}
```

Функция **decode_varint** декодирует значение Base 128 Varint из буфера bufp и возвращает раскодированное значение.

Переменная cur указывает на текущую позицию в буфере bufp. Первый байт в буфере считывается в переменную byte, а младшие 7 битов этого байта сохраняются в переменной value. Затем цикл while, который продолжается до тех пор, пока значение байта byte имеет установленный старший бит (0x80). Внутри цикла считывается следующий байт из буфера bufp в переменную byte и добавляются младшие 7 битов этого байта к переменной value, сдвигая их на shift битов влево. Затем увеличивается значение shift на 7, чтобы готовиться к чтению следующего байта.

После того, как прочитались все байты, необходимые для кодирования значения value, устанавливается указатель bufp на следующую пустую позицию в буфере, где закончилось чтение. Возвращается раскодированное значение value.

```
uint32_t generate_number()
{
    const int r = rand();
    const int p = r % 100;
    if (p < 90)
    {
        return r % 128;
    }
    if (p < 95)
    {
        return r % 16384;
    }
    if (p < 99)
    {
        return r % 2097152;
    }
    return r % 268435455;
}
```

Функция **generate_number** генерирует случайное число.

```
size_t write_bin_file(FILE *file_unc, FILE *file_c)
{
    assert(file_unc != NULL);
    assert(file_c != NULL);

    uint8_t buf[4] = {};
    size_t size = 0;
    size_t full_size = 0;

    for (int i = 0; i < N; i++)
    {
        uint32_t numb = generate_number();
        fwrite(&numb, sizeof(numb), 1, file_unc);
        size = encode_varint(numb, buf);
        full_size += size;
        fwrite(buf, size, 1, file_c);
    }

    return full_size;
}
```

Функция **write_bin_file** записывает сгенерированные числа в два файла: в несжатом формате, в формате varint. Макрос assert проверяет не равны ли файлы NULL. Создаётся массив буфер, для передачи данных в функцию кодирования числа. Создаётся цикл для генерации случайных чисел и запись в файлы сначала без кодирования, а затем с кодированием. Возвращается полный размер закодированных чисел.

```c
// считывать из файла
int read_bin_file(FILE *file_unc, FILE *file_c, size_t size)
{
    assert(file_unc != NULL);
    assert(file_c != NULL);

    fseek(file_c, 0, SEEK_SET);
    fseek(file_unc, 0, SEEK_SET);

    uint32_t num_uncomp, num_comp;
    uint8_t *buf = malloc(sizeof(uint8_t) * size);
    const uint8_t *cur = buf;
    fread(buf, sizeof(uint8_t), size, file_c);

    for (int i = 0; i < N; i++)
    {
        num_comp = decode_varint(&cur);
        fread(&num_uncomp, sizeof(uint32_t), 1, file_unc);
        if (i < 5)
            printf("decode %d encode %d\n", num_comp, num_uncomp);
        if (num_comp != num_uncomp)
            return -1;
    }

    free(buf);
    return 0;
}
```

Функция, которая считывает цифры из файлов. Для начала, проверка с помощью макроса assert равны ли файлы NULL. С помощью fseek указатель перемещается на начало файла. Далее выделяется память для раскодирования чисел. Затем в buf считываются данные из кодированного файла. Числа декодируются и выводится 5 сообщений с кодированным значением и декодируемым. В результате buf очищается.

```c
int main()
{
    srand(time(NULL));

    FILE *file_c;
    if ((file_c = fopen("compressed.dat", "wb+")) == NULL)
    {
        fprintf(stderr, "Can't open compresed.dat\n");
        exit(EXIT_FAILURE);
    }

    FILE *file_unc;
    if ((file_unc = fopen("uncompressed.dat", "wb+")) == NULL)
    {
        fprintf(stderr, "Can't open uncompressed.dat\n");
        exit(EXIT_FAILURE);
    }

    size_t full_size = write_bin_file(file_unc, file_c);

    size_t size = ftell(file_c);
    printf("Size with no encode: %d\nSize with encode: %ld\n",
           N * 4,
           full_size);

    printf("Коэффициент сжатия: %.1f\n\n", (double)(N * 4) / size);

    int test = read_bin_file(file_unc, file_c, size);
    if (test == -1)
        printf("Последовательность чисел неверна\n");
    else
        printf("Последовательность в файлах верна\n");

    fclose(file_c);
    fclose(file_unc);
    return 0;
}
```

В начале функции **main** устанавливается таймер, с помощью функции srand и внутри неё функции time, для того, чтобы когда функция выполнялась время было разное и генерировалось разное число. Далее создан указатель на файл, в котором будет храниться указатель на файл, в котором хранятся закодированные цифры. Если файл равен NULL, то выводится сообщение и программа завершается с оишбкой.

2 Задание

```
int encode(uint32_t code_point, CodeUnits *code_unit)
{
    uint8_t count = 0;
    uint32_t i;

    for (i = code_point; i > 0; i >>= 1)
    {
        count++;
    }

    if (count <= 7)
    {
        code_unit->code[0] = code_point;
        code_unit->length = 1;
        return 0;
    }

    if (count <= 11)
    {
        code_unit->code[0] = 0xc0 | (code_point >> 6);
        code_unit->code[1] = 0x80 | (code_point & 0x3f);
        code_unit->length = 2;
        return 0;
    }

    if (count <= 16)
    {
        code_unit->code[0] = 0xe0 | (code_point >> 12);
        code_unit->code[1] = 0x80 | ((code_point & 0xfc0) >> 6);
        code_unit->code[2] = 0x80 | (code_point & 0x3f);
        code_unit->length = 3;
        return 0;
    }

    if (count <= 21)
    {
        code_unit->code[0] = 0xf0 | (code_point >> 18);
        code_unit->code[1] = 0x80 | ((code_point & 0x3f000) >> 12);
        code_unit->code[2] = 0x80 | ((code_point & 0xfc0) >> 6);
        code_unit->code[3] = 0x80 | (code_point & 0x3f);
        code_unit->length = 4;
        return 0;
    }

    return -1;
}
```

Функция принимает два параметра: число, которое необходимо закодировать, параметр, который будет являться результатом кодирования. С помощью цикла for считывается количество значащих бит кодируемого числа. После этого функция проверяет, сколько байтов потребуется для кодирования символа, и заполняет массив code в code_units соответствующими байтами. Если символ может быть закодирован в один байт, то функция сохраняет этот байт в code_units->code[0] и устанавливает длину code_units->length в 1. Если байт больше чем 1, то с помощью масок и логических операций кодируются остальные байты, заполняется массив и устанавливается соответствующая длина.

```
uint32_t decode(const CodeUnits *code_unit)
{
    uint32_t code_point = 0;

    if ((code_unit->code[0] >> 7) == 0)
    {
        code_point = code_unit->code[0];
    }
    else if (code_unit->code[0] <= 0xdf)
    {
        code_point = ((code_unit->code[0] & 0x1f) << 6 | (code_unit->code[1] & 0x3f));
    }
    else if (code_unit->code[0] <= 0xef)
    {
        code_point = (((code_unit->code[0] & 0xf) << 12) | ((code_unit->code[1] & 0x3f) << 6) | (code_unit->code[2] & 0x3f));
    }
    else if (code_unit->code[0] <= 0xf7)
    {
        code_point = (((code_unit->code[0] & 0xf) << 18) | ((code_unit->code[1] & 0x3f) << 12) | ((code_unit->code[2] & 0x3f) << 6) | (code_unit->code[3] & 0x3f));
    }

    return code_point;
}
```

Функция принимает в качестве параметра закодированное представление числа. Есть 4 условия:

1. Если старший байт, сдвинутый вправо на 7 битов, равен нулю, то это число соответствует шаблону в первой строке таблицы, где количество значащих бит кодируемого числа равно 7. В данном случае переменной code_point присваивается значение старшего байта.
2. Если старший байт меньше, либо равен 0xdf (223), то закодированное число соответствует шаблону во второй строке таблицы, т.к. 0xdf – это 1101 1111 (т.е. максимальное число, которое может соответствовать второму шаблону). В данном случае, сначала с помощью побитового И и сдвига влево на 6 бит избавление от шаблонных битов в старшем байте, затем используется побитовое И для второго байта с числом 0x3f (63), чтобы снова избавиться от шаблонных битов. После этого используется операция побитовое ИЛИ, чтобы «склеить» байты.
3. Если старший байт меньше, либо равен 0xef (239), то закодированное число соответствует шаблону в третьей строке таблицы, т.к. 0xef – это 1110 1111 (т.е. максимальное число, которое соответствовать третьему шаблону). В данном случае сначала с помощью побитового И и сдвигов вправо избавление от шаблонных битов во всех байтах числа. После этого используется операция побитовое ИЛИ, чтобы «склеить» байты.
4. Если старший байт меньше, либо равен 0xf7 (247), то закодированное число соответствует шаблону в четвёртой строке таблицы, т.к. 0xf7 – это 11110111 (т.е. максимальное число, которое может соответствовать четвёртому шаблону).
   Функция возвращает результат декодирования в виде переменной code_point.

```c
int read_next_code_unit(FILE *in, CodeUnits *code_unit)
{
    code_unit->length = 0;
    uint8_t byte;

    while (code_unit->length == 0)
    {
        fread(&byte, sizeof(uint8_t), 1, in);
        if (feof(in))
        {
            return -1;
        }
        if (byte < 0x80)
        {
            code_unit->code[code_unit->length++] = byte;
            return 0;
        }
        if (byte >= 0xc0)
        {
            int i = 3;
            uint8_t buf;
            for (buf = byte >> 4; buf != 1; buf >>= 1, i--)
            {
                if ((buf == 0x3) || (buf == 0x7) || (buf == 0xf))
                {
                    code_unit->code[0] = byte;
                    int j = 0;
                    for (j; j != i; j++)
                    {
                        fread(&byte, sizeof(uint8_t), 1, in);
                        if ((byte >= 0x80) && (byte <= 0xbf))
                        {
                            code_unit->code[j + 1] = byte;
                        }
                        else if (!feof(in))
                        {
                            fseek(in, -1, SEEK_CUR);
                            return -1;
                        }
                        else
                        {
                            return -1;
                        }
                    }
                    code_unit->length = i + 1;
                    break;
                }
            }
        }
    }
    return 0;
}
```

Функция **read_next_code_unit** принимает несколько параметров: поток файла in и указатель на структуру code_unit. Сначала функция fread читает следующий байт из файла и сохраняет его в переменную byte, если достигнут конец файла, функция возвращает -1. if(byte < 0x80) – проверяет, является ли byte однобайтовой кодовой единицей. Если является, то он записывается в последовательность кодовых единиц «code_units», длина последовательности увеличивается на 1 и функция возвращает значение 0. If(byte => 0xc0) – проверяет, является ли byte началом многобайтовой кодовой единицы. Если да, то начинается цикл. Затем начинается цикл, который продолжается до тех пор, пока buf не станет равным единице. Внутри цикла переменная buf сдвигается на один бит вправо, а переменная i уменьшается на единицу.if((buf == 0x3) || (buf == 0x7) || (buf == 0xf)) – проверяет, является ли ‘byte’ началом двухбайтовой, трёхбайтовой или четырёхбайтовой кодовой единицей. Далее записывается первый байт и с помощью ещё одного цикла начинают записываться остальные байты, в 98 строчке проверяется, достигнут ли конец файла. Если нет, то происходит возврат указателя на предыдущий байт с помощью функции fseek,

и функция завершается с ошибкой. Если нет, то выполняются следующие условия. В завершение в 105 строчке устанавливается длина массива code на i+1 и выход из внутреннего цикла.

```
int write_code_unit(FILE *out, const CodeUnits *code_unit)
{
    fwrite(code_unit, sizeof(uint8_t), code_unit->length, out);
    return 0;
}
```

Функция **write_code_unit** записывает последовательность кодовых единиц из структуры CodeUnits в поток out.

```
int encode_file(const char *int_file_name, const char *out_file_name)
{
    uint32_t code_point;
    CodeUnits code_units;

    FILE *in_file = fopen(out_file_name, "wb");
    if (in_file == NULL)
    {
        printf("Failed to read file located at %s\n", int_file_name);
        return -1;
    }

    FILE *out_file = fopen(out_file_name, "wb");
    if (out_file == NULL)
    {
        printf("Failed to create file located at %s\n", out_file_name);
        return -1;
    }

    while (!feof(in_file))
    {
        fscanf(in_file, "%" SCNx32, &code_point);
        encode(code_point, &code_units);
        write_code_unit(out_file, &code_units);
    }

    fclose(in_file);
    fclose(out_file);
    return 0;
}
```

Функция **encode_file** принимает два параметра: путь к входному текстовому файлу и путь к выходному бинарному файлу. Сначала происходит попытка открыть для чтения входной текстовый файл, в случае неудачи, возвращается -1. Если всё успешно, то происходит попытка создать бинарный файл, в случае неудачи, возвращается -1. Если всё прошло успешно, то далее идёт цикл, который будет работать, пока не достигнет концы файла. В цикле из файла считываются числа и записываются в переменную code_point. Далее вызывается функция encode, чтобы закодировать считанные числа. После этого вызывается функция write_code_unit, чтобы записать в выходной бинарный файл закодированные числа. После завершения цикла, закрывается как текстовый фал, так и бинарный файл. В случае успеха, возвращается 0.

```c
int decode_file(const char *in_file_name, const char *out_file_name)
{
    CodeUnits code_units;
    FILE *in_file = fopen(in_file_name, "rb");
    if (in_file == NULL)
    {
        printf("Failed to read file located at %s\n", in_file_name);
        return -1;
    }

    FILE *out_file = fopen(in_file_name, "w");
    if (out_file == NULL)
    {
        printf("Failed to create file located at %s\n", out_file_name);
        return -1;
    }

    while (!feof(in_file))
    {
        if (!read_next_code_unit(in_file, &code_units))
        {
            fprintf(out_file, "%" PRIx32 "\n", decode(&code_units));
        }
    }

    fclose(in_file);
    fclose(out_file);

    return 0;
}
```

Функция **decode_file** принимает два параметра: путь к входному бинарному и путь к выходному текстовому файлу. Сначала попытка открыть для чтения входной бинарный файл, в случае неудачи, возвращается -1. Если всё успешно, то далее попытка создать или открыть уже имеющийся текстовый файл, в случае неудачи возвращается -1. Если всё прошло успешно, то далее идёт цикл, который будет работать, пока не достигнет конца файла. В цикле наблюдается условие, если возвращаемое значение функции read_next_code_unit равно нулю, то записывается в текстовый файл декодированное число. После завершения цикла, бинарный и текстовый файл закрываются. В случае успеха возвращается 0.

```c
#include "coder.h"
#include "command.h"
#include <string.h>

int main(int argc, char *argv[])
{
    if (argc == 4)
    {
        if (strcmp(argv[1], "encode") == 0)
        {
            encode_file(argv[2], argv[3]);
        }
        else if (strcmp(argv[1], "decode") == 0)
        {
            decode_file(argv[2], argv[3]);
        }
        else
        {
            printf("Usage:\n");
            printf("coder encode <in-file-name> <out-file-name>\n");
            printf("coder decode <in-file-name> <out-file-name>\n");
            return -1;
        }
    }
    else
    {
        printf("Usage:\n");
        printf("coder encode <in-file-name> <out-file-name>\n");
        printf("coder decode <in-file-name> <out-file-name>\n");
        return -1;
    }

    return 0;
}
```

В функции **main** проверяется количество введённых аргументов командной строки, затем с помощью функции strcmp проверяется, какое действие хочется совершить: закодировать или декодировать число. В случае неправильного ввода аргументов, main вернёт -1. В случае успеха 0.

ПРИЛОЖЕНИЕ

1 Задание

main.c

```c
#include <assert.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define N 1000000

uint32_t generate_number()
{
    const int r = rand();
    const int p = r % 100;
    if (p < 90)
    {
        return r % 128;
    }
    if (p < 95)
    {
        return r % 16384;
    }
    if (p < 99)
    {
        return r % 2097152;
    }
    return r % 268435455;
}

size_t encode_varint(uint32_t value, uint8_t *buf)
{
    assert(buf != NULL);
    uint8_t *cur = buf;
    while (value >= 0x80)
    {
        const uint8_t byte = (value & 0x7f) | 0x80;
        *cur = byte;
        value >>= 7;
        ++cur;
    }
    *cur = value;
    ++cur;
    return cur - buf;
}

uint32_t decode_varint(const uint8_t **bufp)
{
    const uint8_t *cur = *bufp;
    uint8_t byte = *cur++;
    uint32_t value = byte & 0x7f;
    size_t shift = 7;
    while (byte >= 0x80)
    {
        byte = *cur++;
        value += (byte & 0x7f) << shift;
```

```
51          shift += 7;
52      }
53      *bufp = cur;
54      return value;
55  }
56
57  // записывать в файл
585 size_t write_bin_file(FILE *file_unc, FILE *file_c)
9   {
60      assert(file_unc != NULL);
61      assert(file_c != NULL);
62
63      uint8_t buf[4] = {};
64      size_t size = 0;
65      size_t full_size = 0;
66
67      for (int i = 0; i < N; i++)
68      {
69          uint32_t numb = generate_number();
70          fwrite(&numb, sizeof(numb), 1, file_unc);
71          size = encode_varint(numb, buf);
72          full_size += size;
73          fwrite(buf, size, 1, file_c);
74      }
75
76      return full_size;
77  }
78
79  // считывать из файла
80  int read_bin_file(FILE *file_unc, FILE *file_c, size_t size)
81  {
82      assert(file_unc != NULL);
83      assert(file_c != NULL);
84
85      fseek(file_c, 0, SEEK_SET);
86      fseek(file_unc, 0, SEEK_SET);
87
88      uint32_t num_uncomp, num_comp;
89      uint8_t *buf = malloc(sizeof(uint8_t) * size);
90      const uint8_t *cur = buf;
91      fread(buf, sizeof(uint8_t), size, file_c);
92
93      for (int i = 0; i < N; i++)
94      {
95          num_comp = decode_varint(&cur);
96          fread(&num_uncomp, sizeof(uint32_t), 1, file_unc);
97          if (i < 5)
98              printf("decode %d encode %d\n", num_comp, num_uncomp);
99          if (num_comp != num_uncomp)
100             return -1;
101     }
102
103     free(buf);
104     return 0;
105 }
106
107 int main()
108 {
```

```
    srand(time(NULL));

    FILE *file_c;
    if ((file_c = fopen("compressed.dat", "wb+")) == NULL)
    {
        fprintf(stderr, "Can't open compresed.dat\n");
        exit(EXIT_FAILURE);
    }

    FILE *file_unc;
    if ((file_unc = fopen("uncompressed.dat", "wb+")) == NULL)
    {
        fprintf(stderr, "Can't open uncompressed.dat\n");
        exit(EXIT_FAILURE);
    }

    size_t full_size = write_bin_file(file_unc, file_c);

    size_t size = ftell(file_c);
    printf("Size with no encode: %d\nSize with encode: %ld\n",
           N * 4,
           full_size);

    printf("Коэффициент сжатия: %.1f\n\n", (double)(N * 4) / size);

    int test = read_bin_file(file_unc, file_c, size);
    if (test == -1)
        printf("Последовательность чисел неверна\n");
    else
        printf("Последовательность в файлах верна\n");

    fclose(file_c);
    fclose(file_unc);
    return 0;
}
```

2 Задание

coder.c

```
#include <stdio.h>
#include "coder.h"

uint32_t decode(const CodeUnits *code_unit)
{
    uint32_t code_point = 0;

    if ((code_unit->code[0] >> 7) == 0)
    {
        code_point = code_unit->code[0];
    }
    else if (code_unit->code[0] <= 0xdf)
```

```
    {
        code_point = ((code_unit->code[0] & 0x1f) << 6 | (code_unit->code[1] & 0x3f));
    }
    else if (code_unit->code[0] <= 0xef)
    {
        code_point = (((code_unit->code[0] & 0xf) << 12) | ((code_unit->code[1] & 0x3f) << 6) | (code_unit->code[2] & 0x3f));
    }
    else if (code_unit->code[0] <= 0xf7)
    {
        code_point = (((code_unit->code[0] & 0xf) << 18) | ((code_unit->code[1] & 0x3f) << 12) | ((code_unit->code[2] & 0x3f) << 6) | (code_unit->code[3] & 0x3f));
    }

    return code_point;
}

int encode(uint32_t code_point, CodeUnits *code_unit)
{
    uint8_t count = 0;
    uint32_t i;

    for (i = code_point; i > 0; i >>= 1)
    {
        count++;
    }

    if (count <= 7)
    {
        code_unit->code[0] = code_point;
        code_unit->length = 1;
        return 0;
    }

    if (count <= 11)
    {
        code_unit->code[0] = 0xc0 | (code_point >> 6);
        code_unit->code[1] = 0x80 | (code_point & 0x3f);
        code_unit->length = 2;
        return 0;
    }

    if (count <= 16)
    {
        code_unit->code[0] = 0xe0 | (code_point >> 12);
        code_unit->code[1] = 0x80 | ((code_point & 0xfc0) >> 6);
        code_unit->code[2] = 0x80 | (code_point & 0x3f);
        code_unit->length = 3;
```

```
        return 0;
    }

    if (count <= 21)
    {
        code_unit->code[0] = 0xf0 | (code_point >> 18);
        code_unit->code[1] = 0x80 | ((code_point & 0x3f000) >> 12);
        code_unit->code[2] = 0x80 | ((code_point & 0xfc0) >> 6);
        code_unit->code[3] = 0x80 | (code_point & 0x3f);
        code_unit->length = 4;
        return 0;
    }

    return -1;
}

int write_code_unit(FILE *out, const CodeUnits *code_unit)
{
    fwrite(code_unit, sizeof(uint8_t), code_unit->length, out);
    return 0;
}

int read_next_code_unit(FILE *in, CodeUnits *code_unit)
{
    code_unit->length = 0;
    uint8_t byte;

    while (code_unit->length == 0)
    {
        fread(&byte, sizeof(uint8_t), 1, in);
        if (feof(in))
        {
            return -1;
        }
        if (byte < 0x80)
        {
            code_unit->code[code_unit->length++] = byte;
            return 0;
        }
        if (byte >= 0xc0)
        {
            int i = 3;
            uint8_t buf;
            for (buf = byte >> 4; buf != 1; buf >>= 1, i--)
            {
                if ((buf == 0x3) || (buf == 0x7) || (buf == 0xf))
                {
                    code_unit->code[0] = byte;
                    int j = 0;
                    for (j; j != i; j++)
```

```
                    {
                        fread(&byte, sizeof(uint8_t), 1, in);
                        if ((byte >= 0x80) && (byte <= 0xbf))
                        {
                            code_unit->code[j + 1] = byte;
                        }
                        else if (!feof(in))
                        {
                            fseek(in, -1, SEEK_CUR);
                            return -1;
                        }
                        else
                        {
                            return -1;
                        }
                    }
                    code_unit->length = i + 1;
                    break;
                }
            }
        }
    }
    return 0;
}
```

coder.h

```
#pragma once
#include <stdio.h>
#include <stdint.h>

enum
{
    MaxCodeLength = 4
};

typedef struct
{
    uint8_t code[MaxCodeLength];
    size_t length;
} CodeUnits;

int encode(uint32_t code_point, CodeUnits *code_units);

uint32_t decode(const CodeUnits *code_unit);

int read_next_code_unit(FILE *in, CodeUnits *code_units);
```

```
int write_code_unit(FILE *out, const CodeUnits *code_unit);
```

command.c

```
#include "coder.h"
#include "command.h"
#include <inttypes.h>
#include <stdio.h>

int encode_file(const char *int_file_name, const char *out_file_name)
{
    uint32_t code_point;
    CodeUnits code_units;

    FILE *in_file = fopen(out_file_name, "wb");
    if (in_file == NULL)
    {
        printf("Failed to read file located at %s\n", int_file_name);
        return -1;
    }

    FILE *out_file = fopen(out_file_name, "wb");
    if (out_file == NULL)
    {
        printf("Failed to create file located at %s\n", out_file_name);
        return -1;
    }

    while (!feof(in_file))
    {
        fscanf(in_file, "%" SCNx32, &code_point);
        encode(code_point, &code_units);
        write_code_unit(out_file, &code_units);
    }

    fclose(in_file);
    fclose(out_file);
    return 0;
}

int decode_file(const char *in_file_name, const char *out_file_name)
{
    CodeUnits code_units;
    FILE *in_file = fopen(in_file_name, "rb");
    if (in_file == NULL)
    {
        printf("Failed to read file located at %s\n", in_file_name);
```

```
        return -1;
    }

    FILE *out_file = fopen(in_file_name, "w");
    if (out_file == NULL)
    {
        printf("Failed to create file located at %s\n", out_file_name);
        return -1;
    }

    while (!feof(in_file))
    {
        if (!read_next_code_unit(in_file, &code_units))
        {
            fprintf(out_file, "%" PRIx32 "\n", decode(&code_units));
        }
    }

    fclose(in_file);
    fclose(out_file);

    return 0;
}
```

command.h

```
#pragma once

int encode_file(const char *int_file_name, const char *out_file_name);
int decode_file(const char *in_file_name, const char *out_file_name);
```

main.c

```
#include "coder.h"
#include "command.h"
#include <string.h>

int main(int argc, char *argv[])
{
    if (argc == 4)
    {
        if (strcmp(argv[1], "encode") == 0)
        {
            encode_file(argv[2], argv[3]);
```

```
        }
        else if (strcmp(argv[1], "decode") == 0)
        {
            decode_file(argv[2], argv[3]);
        }
        else
        {
            printf("Usage:\n");
            printf("coder encode <in-file-name> <out-file-name>\n");
            printf("coder decode <in-file-name> <out-file-name>\n");
            return -1;
        }
    }
    else
    {
        printf("Usage:\n");
        printf("coder encode <in-file-name> <out-file-name>\n");
        printf("coder decode <in-file-name> <out-file-name>\n");
        return -1;
    }

    return 0;
}
```