



Министерство науки и высшего образования Российской Федерации
Калужский филиал
федерального государственного бюджетного
образовательного учреждения высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИУК «Информатика и управление»**

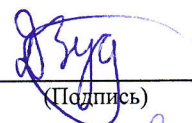
КАФЕДРА **ИУК4 «Программное обеспечение ЭВМ, информационные технологии»**

ЛАБОРАТОРНАЯ РАБОТА №5

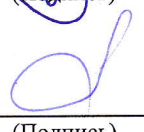
«Обработка бинарных файлов»

ДИСЦИПЛИНА: «Типы и структуры данных»

Выполнил: студент гр. ИУК4-32Б

 (____ Зудин Д.В. ____)
(Подпись) (Ф.И.О.)

Проверил:

 (____ Пчелинцева Н.И. ____)
(Подпись) (Ф.И.О.)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка: 

- Оценка: 

Калуга, 2022 г.

Цель: формирование практических навыков создания алгоритмов обработки бинарных файлов.

Задачи:

1. Познакомиться со структурой бинарного bmp-файла;
2. Изучить способы программной обработки бинарного файла;
3. Реализовать алгоритм согласно варианту.

Вариант №3

Формулировка задания

1. Обработать/создать/изменить файл, содержание которого предусмотрено вариантом задания.
2. Все приложение в целом должно быть написано с использованием ООП-технологии.
3. Все входные и выходные данные считать/записать из/в BMP-файл(а).
4. Все возникшие ошибки записать в файл ошибок.
5. В консольном приложении продемонстрировать работу программы.
6. Программа должна запускаться из командной строки с указанием имени исполняемого файла, имен файлов входных, выходных данных, файла ошибок.
7. Предоставить и защитить отчет.

Индивидуальное задание

Дан BMP-файл, содержащий рисунок. Необходимо увеличить размер рисунка в два раза.

Листинг файла FileLogging.h

```
#ifndef FILE_LOGGING
#define FILE_LOGGING
#include <string>
#include <fstream>
#include <ctime>
#include <iostream>

class FileLogging
{
public:
    FileLogging(std::string fileName);
    void Logging(std::string message);

private:
    std::string getTime();
    std::string fileName;
};
#endif
```

Листинг файла FileLogging.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include "FileLogging.h"

FileLogging::FileLogging(std::string fileName)
{
    this->fileName = fileName;
}

void FileLogging::Logging(std::string message)
{
    std::ofstream fout(fileName, std::ios::out | std::ios::app);
    if (fout.is_open())
    {
        fout << "[" << getTime() << "]" " << message << "\n";
    }
    fout.close();
}

std::string FileLogging::getTime()
{
    time_t seconds = time(nullptr);
    tm* timeinfo = localtime(&seconds);
    std::string currTime = asctime(timeinfo);
    currTime.pop_back();
    return currTime;
}
```

Листинг файла BMP.h

```
#ifndef BMP_H
#define BMP_H

#include <iostream>
#include <cstdlib>
#include <fstream>
#include <vector>
#include <string>

class BMP
{
public:
    // Конструктор с параметрами. Загружает растровое изображение с диска
    // @param file_path путь к файлу изображения с расширением bmp
    explicit BMP(const std::string& file_path);

    // Конструктор с параметрами. Создает растровое изображение
    // @param width - ширина растрового изображения
    // @param height - высота растрового изображения
    // @param has_alpha - наличие канала прозрачности
    BMP(int32_t width, int32_t height, bool has_alpha = true);

    // Загрузка растрового изображения с диска
    // @param file_path - путь для загрузки
    void load(const std::string& file_path);

    // Сохранение растрового изображения на диск
    // @param file_path - путь для записи
    void save(const std::string& file_path);

    // Заполнение указанной области указанным цветом

```

```

// @param x0 - координата для начала отсчета по оси x
// @param y0 - координата для начала отсчета по оси y
// @param width - ширина заполняемой области
// @param height - высота заполняемой области
// @param R - значение канала красного
// @param G - значение канала зеленого
// @param B - значение канала синего
// @param A - значение канала прозрачности
void fillRegion(uint32_t x0, uint32_t y0, uint32_t width, uint32_t
height, uint8_t R,
                uint8_t G, uint8_t B, uint8_t A);

// Масштабирование растрового изображения
// @param new_width - ширина, до которой нужно произвести
масштабирование
// @param new_height - высота, до которой нужно произвести
масштабирование
void scale(int32_t new_width, int32_t new_height);

// Получение ширины растрового изображения
// @return - текущая ширина изображения
int32_t getWidth() const;

// Получение высоты растрового изображения
// @return - текущая высота растрового изображения
int32_t getHeight() const;

// Получение негатива
// @param file_path - путь для записи
void getNegative(const std::string& file_path);

protected:
#pragma pack(push, 1)
    struct FileHeader
    {
        // Отметка для отличия формата от других (сигнатура формата).
        // Может содержать
        // единственное значение 0x4D42
        uint16_t file_type{0x4D42};
        // Размер файла в байтах
        uint32_t file_size{};
        // Зарезервированное поле. Всегда должно содержать ноль
        uint16_t reserved1{};
        // Зарезервированное поле. Всегда должно содержать ноль
        uint16_t reserved2{};
        // Положение пиксельных данных относительно начала данной
        структуры
        uint32_t offset_data{};
    };
#pragma pack(pop)

    struct InfoHeader
    {
        // Размер данной структуры в байтах, указывающий также на версию
        структуры
        uint32_t size{};
        // Ширина растрового изображения в пикселях
        int32_t width{};
        // Высота растрового изображения в пикселях
        int32_t height{};
        // Количество цветовых плоскостей. Всегда должно быть равно
        единице
        uint16_t planes{1};
    };

```

```

        // Количество бит на пиксель
        uint16_t bit_count{};
        // Способ хранения пикселей
        uint32_t compression{};
        // Размер пиксельных данных в байтах
        uint32_t size_image{};
        // Количество пикселей на метр по горизонтали
        int32_t x_pixels_per_meter{};
        // Количество пикселей на метр по вертикали
        int32_t y_pixels_per_meter{};
        // Количество цветов в цветовой палитре
        uint32_t colors_used{};
        // Цвета, используемые растровым изображением
        uint32_t colors_important{};
    };

    struct ColorHeader
    {
        // Битовая маска для канала красного
        uint32_t red_mask{ 0x00ff0000 };
        // Битовая маска для канала зеленого
        uint32_t green_mask{ 0x0000ff00 };
        // Битовая маска для канала синего
        uint32_t blue_mask{ 0x000000ff };
        // Битовая маска для альфа-канала
        uint32_t alpha_mask{ 0xff000000 };
        // Тип цветового пространства. По умолчанию sRGB (0x73524742)
        uint32_t color_space_type{ 0x73524742 };
        // Неиспользуемые данные для цветового пространства sRGB
        uint32_t unused[16]{};
    };

    // Заголовочная структура текущего растрового изображения
    FileHeader m_file_header;
    // Информационная структура текущего растрового изображения
    InfoHeader m_info_header;
    // Цветовая структура текущего растрового изображения
    ColorHeader m_color_header;
    // Хранилище для информации о пикселях
    std::vector<uint8_t> m_data;
    // Количество значимых байт
    uint32_t m_row_stride;

    // Проверка формата цвета на соответствие форматам цветов
    // @param color_header - ColorHeader с заданными битовыми масками
    // и типом цветового пространства
    void _checkColorHeader(ColorHeader& color_header);
    // Выравнивание длины строки
    // @param align_stride - коэффициент выравнивания
    // @return - длина выравненной строки
    uint32_t _makeStrideAligned(uint32_t align_stride) const;
    // Запись заголовков на диск
    // @param stream - поток для записи
    void _writeHeaders(std::ofstream& stream);
    // Запись заголовков и информации о пикселях на диск
    // @param stream - поток для записи
    void _writeHeadersAndData(std::ofstream& stream);
};
#endif

```

Листинг файла BMP.cpp

```
#include "BMP.h"

BMP::BMP(const std::string& file_path) : m_row_stride{ 0 }
{
    load(file_path);
}

BMP::BMP(int32_t width, int32_t height, bool has_alpha)
{
    // проверяем заданную ширину и высоту
    if (width <= 0 || height <= 0)
    {
        throw std::runtime_error("The image width and height must be positive numbers.");
    }
    m_info_header.width = width;
    m_info_header.height = height;
    // если задано 4 канала (32-битное изображение)
    if (has_alpha)
    {
        // записываем размер информационного хедера (InfoHeader и ColorHeader относятся к информационному хедеру)
        m_info_header.size = sizeof(InfoHeader) + sizeof(ColorHeader);
        // записываем, через сколько байт от начала файла начинаются пиксели
        m_file_header.offset_data = sizeof(FileHeader) + sizeof(InfoHeader) + sizeof(ColorHeader);
        // записываем, что на пиксель приходится 32 бита информации
        m_info_header.bit_count = 32;
        // записываем, что изображение не сжатое
        m_info_header.compression = 3;
        // вычисляем количество значимых байт
        m_row_stride = width * 4;
        // выделяем память для хранения информации о пикселях (ширина * высота * количество каналов)
        m_data.resize(m_row_stride * height);
        // записываем размер файла (служебная информация + информация о пикселях)
        m_file_header.file_size = m_file_header.offset_data + m_data.size();
    }
    // если задано 3 канала (24-битное изображение)
    else
    {
        // записываем размер информационного хедера
        m_info_header.size = sizeof(InfoHeader);
        // записываем, через сколько байт от начала файла начинаются пиксели
        m_file_header.offset_data = sizeof(FileHeader) + sizeof(InfoHeader);
        // записываем битность изображения
        m_info_header.bit_count = 24;
        // записываем способ хранения информации о пикселях
        m_info_header.compression = 0;
        // записываем количество значимых байт
        m_row_stride = width * 3;
        // выделяем память для информации о пикселях
        m_data.resize(m_row_stride * height);
        // вычисляем длину строки для выравнивания
        uint32_t new_stride{ _makeStrideAligned(4) };
        // записываем размер файла
        m_file_header.file_size = m_file_header.offset_data + m_data.size() + m_info_header.height * (new_stride - m_row_stride);
    }
}
```

```

void BMP::load(const std::string& file_path)
{
    std::ifstream input{ file_path, std::ios::binary };
    if (!input)
    {
        throw std::runtime_error("Unable to open the input image file!");
    }
    // считываем заголовочную структуру изображения
    input.read((char*)&m_file_header, sizeof(m_file_header));
    // проверяем формат изображения
    if (m_file_header.file_type != 0x4D42)
    {
        throw std::runtime_error("Unrecognized file format!");
    }
    // считываем информационный хэдер
    input.read((char*)&m_info_header, sizeof(m_info_header));
    // если изображение 32-битное, нужно проверить и считать ColorHeader
    if (m_info_header.bit_count == 32)
    {
        // проверяем содержит ли файл ColorHeader (записывается только в 32-
        битные изображения)
        if (m_info_header.size >= (sizeof(InfoHeader) + sizeof(ColorHeader)))
        {
            // считываем ColorHeader
            input.read((char*)&m_color_header, sizeof(m_color_header));
            // проверяем, записаны ли пиксели в формате BGRA и является ли
            цветовое пространство sRGB
            _checkColorHeader(m_color_header);
        }
        // если не содержит - выбрасываем исключение
        else
        {
            std::cerr << "Warning! The file \"" << file_path << "\" does not
            seem to contain bit mask information\n";
            throw std::runtime_error("Error! Unrecognized file format!");
        }
    }
    // перемещаемся к началу пикселей
    input.seekg(m_file_header.offset_data, std::ifstream::beg);
    // в силу того, что некоторые фоторедакторы помещают в файл служебную
    информацию, которую можно спокойно игнорировать,
    // чтобы ничего не сбилось, нужно настроить размер информационного
    хедера, размер файла и указать, откуда начинаются пиксели
    // если изображение 32-битное
    if (m_info_header.bit_count == 32)
    {
        // записываем размер информационного хедера, учитывая ColorHeader
        m_info_header.size = sizeof(InfoHeader) + sizeof(ColorHeader);
        // записываем, через сколько байт от начала файла начинаются пиксели,
        учитывая ColorHeader
        m_file_header.offset_data = sizeof(FileHeader) + sizeof(InfoHeader) +
        sizeof(ColorHeader);
    }
    // если изображение 24-битное
    else
    {
        // записываем размер информационного хедера
        m_info_header.size = sizeof(InfoHeader);
        // записываем, через сколько байт от начала файла начинаются пиксели
        m_file_header.offset_data = sizeof(FileHeader) + sizeof(InfoHeader);
    }
    // записываем размер файла (это не весь размер файла, а лишь то,

```

```

// сколько занимают заголовочные структуры и служебная информация)
m_file_header.file_size = m_file_header.offset_data;
// если высота изображения задается отрицательным числом,
// то отсчет пикселей ведется сверху-вниз, начиная в верхнем левом углу

// если высота изображения задается положительным числом,
// то отсчет пикселей ведется снизу-вверх, начиная в нижнем левом углу

// данная программа берет за начало отсчета нижний левый угол

// проверяю высоту
if (m_info_header.height < 0)
{
    throw std::runtime_error("The program can treat only BMP images with
the origin in the bottom left corner!");
}
// выделяю память для хранения информации о пикселях (ширина * высота *
количество каналов)
m_data.resize(m_info_header.width * m_info_header.height *
m_info_header.bit_count / 8);

// Формат изображения BMP предполагает, что каждая строка данных будет
выровнена по границе четырех байтов или
// дополнена нулями, если это не так. Для изображения с разрешением 32
бита на пиксель условие выравнивания всегда выполняется.
// В случае изображений с разрешением 24 бита на пиксель условие
выравнивания выполняется только в том случае,
// если ширина изображения делится на 4, в противном случае нам нужно
будет заполнить строки нулями.

// если ширина изображения делится на 4
if (m_info_header.width % 4 == 0)
{
    // считываем информацию о пикселях
    input.read((char*)m_data.data(), m_data.size());
    // обновляем размер файла (добавляем к нему то, сколько занимает
информация о пикселях)
    m_file_header.file_size += m_data.size();
}
// если ширина изображения не делится на 4
else
{
    // вычисляем width
    // (количество пикселей в строке * количество каналов = количество
байт в строке (width))
    m_row_stride = m_info_header.width * m_info_header.bit_count / 8;

    // вычисляем line_stride
    uint32_t new_stride{ _makeStrideAligned(4) };
    // выделяем память для отступа (line_padding)
    std::vector<uint8_t> padding_row(new_stride - m_row_stride);

    // проходимся по каждой строке пикселей
    for (int y = 0; y < m_info_header.height; y++)
    {
        // считываем информацию об отступе в информацию о пикселях
        // m_data.data() - начало отсчета
        // m_row_stride - сколько нужно отсупить, чтобы попасть на начало
отступа
        // y - поскольку в памяти компьютера информация хранится в виде
последовательности байт (как одномерный массив),
        // смена строки пикселей производится через умножение
        // изначально имеем пустой вектор m_data

```



```

        // сперва считываем в него информацию о имеющихся пикселях
        input.read((char*)(m_data.data() + m_row_stride * y),
m_row_stride);
        // есть два варианта, как поступить с отступом:
        // 1) считать незначащие нули
        // 2) программно переместить указатель на следующую строку
        // потом считываем отступ
        input.read((char*)padding_row.data(), padding_row.size());
    }
    // обновляем размер файла (размер пикселей + высота изображения *
размер отступа)
    m_file_header.file_size += m_data.size() + m_info_header.height *
padding_row.size();
}
}

void BMP::getNegative(const std::string& new_file_path)
{
    // вычисляем количество каналов (глубина цвета / 8)
    uint32_t channels = m_info_header.bit_count / 8;
    // проходимся по пикселям заданной области
    for (uint32_t y = 0; y < m_info_header.height; y++)
    {
        for (uint32_t x = 0; x < m_info_header.width; x++)
        {
            // задаем цвет каждого отдельного пикселя в формате BGRA
            for (int i = 0; i < 3; i++)
            {
                m_data.at(channels * (y * m_info_header.width + x) + i) = 255
- m_data.at(channels * (y * m_info_header.width + x) + i);
            }
            // если имеем 4 канала (32-битное изображение)
            if (channels == 4)
            {
                // задаем компонент прозрачности
                m_data.at(channels * (y * m_info_header.width + x) + 3) =
m_data.at(channels * (y * m_info_header.width + x) + 3);
            }
        }
    }
    save(new_file_path);
}

void BMP::save(const std::string& file_path)
{
    // открываем файловый поток в бинарном режиме
    std::ofstream output{ file_path, std::ios::binary };
    // проверяем, открылся ли файл
    if (!output)
    {
        throw std::runtime_error("Unable to open the input image file!");
    }
    // если изображение 32-битное
    if (m_info_header.bit_count == 32)
    {
        _writeHeadersAndData(output);
    }
    // если изображение 24-битное
    else if (m_info_header.bit_count == 24)
    {
        // если ширина изображения делится на 4
        if (m_info_header.width % 4 == 0)
        {

```

```

        // записываем всю информацию об изображении
        _writeHeadersAndData(output);
    }
    // если ширина изображения не делится на 4
    else
    {
        // вычисляем line_stride
        uint32_t new_stride{ _makeStrideAligned(4) };
        // выделяем память для отступа (line_padding)
        std::vector<uint8_t> padding_row(new_stride - m_row_stride);
        // записываем только хэдеры
        _writeHeaders(output);
        for (int y = 0; y < m_info_header.height; y++)
        {
            // записываем информацию о пикселях
            output.write((const char*)(m_data.data() + m_row_stride * y),
m_row_stride);
            // записываем отступ
            output.write((const char*)padding_row.data(),
padding_row.size());
        }
    }
    // выбрасываем исключение (данная программа обрабатывает только 24- или
32-битные изображения)
    else
    {
        throw std::runtime_error("The program can treat only 24 or 32 bits
per pixel BMP files");
    }
}

void BMP::fillRegion(uint32_t x0, uint32_t y0, uint32_t width, uint32_t
height, uint8_t R, uint8_t G, uint8_t B, uint8_t A)
{
    // проверяем полученные данные, чтобы они соответствовали текущему
изображению
    if (x0 + width > (uint32_t)m_info_header.width || y0 + height >
(uint32_t)m_info_header.height)
    {
        throw std::runtime_error("The region does not fit in the image!");
    }
    // вычисляем количество каналов (глубина цвета / 8)
    uint32_t channels = m_info_header.bit_count / 8;
    // проходимся по пикселям заданной области
    for (uint32_t y = y0; y < y0 + height; ++y)
    {
        for (uint32_t x = x0; x < x0 + width; ++x)
        {
            // задаем цвет каждого отдельного пикселя в формате BGRA
            m_data.at(channels * (y * m_info_header.width + x) + 0) = B;
            m_data.at(channels * (y * m_info_header.width + x) + 1) = G;
            m_data.at(channels * (y * m_info_header.width + x) + 2) = R;
            // если имеем 4 канала (32-битное изображение)
            if (channels == 4)
            {
                // задаем компонент прозрачности
                m_data.at(channels * (y * m_info_header.width + x) + 3) = A;
            }
        }
    }
}

```

```

void BMP::scale(int32_t new_width, int32_t new_height)
{
    // вычисляем соотношение ширины исходного изображения к новой ширине
    uint32_t x_ratio = ((m_info_header.width << 16) / new_width) + 1;
    // вычисляем соотношение высоты исходного изображения к новой высоте
    uint32_t y_ratio = ((m_info_header.height << 16) / new_height) + 1;
    // вычисляем количество каналов
    uint32_t channels = m_info_header.bit_count / 8;
    // выделяем память для информации о пикселях
    std::vector<uint8_t> temp(new_width * new_height * channels);
    uint32_t x_2{}, y_2{};
    for (uint32_t i = 0; i < new_height; ++i)
    {
        for (uint32_t j = 0; j < new_width; ++j)
        {
            // вычисляем позицию соседнего пикселя по оси x
            x_2 = ((j * x_ratio) >> 16);
            // вычисляем позицию соседнего пикселя по оси y
            y_2 = ((i * y_ratio) >> 16);
            // переносим необходимые пиксели в новый вектор
            for (int k = 0; k < 3; k++)
            {
                temp[channels * (i * new_width + j) + k] = m_data[channels *
(y_2 * m_info_header.width + x_2) + k];
            }
            if (channels == 4)
            {
                temp[channels * (i * new_width + j) + 3] = m_data[channels *
(y_2 * m_info_header.width + x_2) + 3];
            }
        }
    }
    // обновляем информацию об изображении
    m_info_header.width = new_width;
    m_info_header.height = new_height;
    m_data = temp;
}

void BMP::_checkColorHeader(ColorHeader& color_header)
{
    ColorHeader expected_color_header{};
    // проверяем формат цвета, чтобы он соответствовал BGRA формату
    if (expected_color_header.red_mask != color_header.red_mask ||
        expected_color_header.blue_mask != color_header.blue_mask ||
        expected_color_header.green_mask != color_header.green_mask ||
        expected_color_header.alpha_mask != color_header.alpha_mask)
    {
        throw std::runtime_error("Unexpected color mask format! "
            "The program expects the pixel data to be in the BGRA format!");
    }
    // проверяем цветовое пространство, чтобы оно соответствовало sRGB
    if (expected_color_header.color_space_type !=
color_header.color_space_type)
    {
        throw std::runtime_error("Unexpected color space type! The program
expects sRGB values");
    }
}

uint32_t BMP::_makeStrideAligned(uint32_t align_stride) const
{
    // берем текущее количество байт в строке
    uint32_t new_stride{ m_row_stride };

```

```

        // увеличиваем до тех пор, пока не будет делиться на 4
        while (new_stride % align_stride != 0)
        {
            ++new_stride;
        }
        return new_stride;
    }

void BMP::_writeHeaders(std::ofstream& stream)
{
    // записываем в файл заголовочную структуру изображения
    stream.write((const char*)&m_file_header, sizeof(m_file_header));
    // записываем в файл информационную структуру изображения
    stream.write((const char*)&m_info_header, sizeof(m_info_header));
    // если изображение 32-битное
    if (m_info_header.bit_count == 32)
    {
        // записываем в файл ColorHeader
        stream.write((const char*)&m_color_header, sizeof(m_color_header));
    }
}

void BMP::_writeHeadersAndData(std::ofstream& stream)
{
    // записываем хэдеры
    _writeHeaders(stream);
    // записываем в файл информацию о пикселях
    stream.write((const char*)m_data.data(), m_data.size());
}

int32_t BMP::getWidth() const
{
    return m_info_header.width;
}

int32_t BMP::getHeight() const
{
    return m_info_header.height;
}

```

Листинг файла main.cpp

```

#include "BMP/BMP.h"
#include "FileLogging/FileLogging.h"

int main(int argc, char* argv[])
{
    setlocale(LC_ALL, "Russian");
    std::string bmp_file{};
    std::string new_bmp_file{};
    std::string error_file{ "error_log.txt" };
    if (argc == 2)
    {
        bmp_file = new_bmp_file = argv[1];
    }
    else if (argc == 3)
    {
        bmp_file = argv[1];
        new_bmp_file = argv[2];
    }
    else if (argc == 4)
    {

```


Вывод информации на консоль

```
Изначальная ширина изображения: 1920
Изначальная высота изображения: 1080

Изменённая ширина изображения: 3840
Изменённая высота изображения: 2160
```

Конечные размеры растрового изображения

Изменить размер

☒ Пиксели ☐ Проценты

Ширина: 3840 Высота: 2160

Качество: 100% (Высокое) Расширение файла: .bmp

Текущее:	3840 x 2160 пикселей	31.6 МБ	.bmp
Новое:	3840 x 2160 пикселей	31.6 МБ	.bmp

Сохранить Отмена

Содержимое файла ошибок при их возникновении в программе

```
[Wed Dec 28 21:45:54 2022] Unable to open the input image file!
```

Выводы:

В ходе работы были сформированы практические навыки создания алгоритмов обработки бинарных файлов.