|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **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 <cstdint>  #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 х  // @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 0х4D42  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 ьекущего 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 хэдера (InfoHeader и ColorHeader относятся к информационному хэдеру)  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;  // выделям 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 хэдера  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;  // записывем 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) };  // записыаем 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)  {  // проверям 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 хэдера, размер файла и указать, откуда начинаются пиксели  // если изображение 32-битное  if (m\_info\_header.bit\_count == 32)  {  // записываем размер информационного хэдера, 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 хэдера  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  // проверям 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!");  }  // выделям 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)  {  // ели ширина изображения делится на 4  if (m\_info\_header.width % 4 == 0)  {  // записываем всю информацию об изображении  \_writeHeadersAndData(output);  }  // ели 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++)  {  // записывем 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)  {  bmp\_file = argv[1];  new\_bmp\_file = argv[2];  error\_file = argv[3];  }  FileLogging error\_log(error\_file);  try  {  BMP bmp(bmp\_file);  std::cout << "Изначальная ширина изображения: " << bmp.getWidth() << "\n";  std::cout << "Изначальная высота изображения: " << bmp.getHeight() << "\n";  bmp.scale(2 \* bmp.getWidth(), 2 \* bmp.getHeight());  std::cout << "\nИзменённая ширина изображения: " << bmp.getWidth() << "\n";  std::cout << "Изменённая высота изображения: " << bmp.getHeight() << "\n";  bmp.save(new\_bmp\_file);  }  catch (const std::exception& e)  {  error\_log.Logging(e.what());  }  return 0;  }  **Результат выполнения программы**  **Запуск программы из консоли**    **Изначальные размеры растрового изображения**    **Вывод информации на консоль**    **Конечные размеры растрового изображения**    **Содержимое файла ошибок при их возникновении в программе**    **Выводы:**  В ходе работы были сформированы практические навыки создания алгоритмов обработки бинарных файлов. | | |
|  | | |