Homework 2

Image processing is an important program for scientific computing. How to improve the computation performance of image processing is the key issue of the program design. In the homework, you must to study some technologies by yourself to improve the computation performance of image processing. The program attached in the document is used to do image processing. There are 3 rar files listed in the e-learn system. The first rar file contains 11 test files, the second rar file contains 7 files, and the last rar file contains 2 files. The total test files are 20. The testing OS is Ubuntu 24.04.1.

In the first, you should compile the codes attached in the file (as the original program), execute it on your computer, and get the computing time. The computing time of the original program is the bench mark. The computing time of the original program on teacher's environment with the 20 test files is 14.7829, and The computing time of the multi-thread program on the test computer is 4.0808.

Then, you can improve your program from the program attached in the file, compile it and execute it on your computer. You can use the pthread or C++ 11 thread to modify your program.

Pthread: https://man7.org/linux/man-pages/man3/pthread_create.3.html

C++ 11 thread: https://en.cppreference.com/w/cpp/thread/thread.html

Assume that the computing time of the original program on your computer is a, and the computing time of your modified program on your computer is b. The b should be less than a. Suppose b is larger than or equal to a, the grade of your homework 2 will be very low.

You must upload your source codes (.c or .cpp) and some usage documents (for example, how to compile and execute your program) to elearn platform. Your source codes will be compiled and executed on the teacher’s computer. The grade of your homework 2 is generated according to the computing time of your program. Assume that the computing time of the original program on teacher’s environment with 100 test files is X seconds, and computing time of the multi-thread program on the test computer with 100 test files is Y seconds.

1. If the computing time of your program is larger than X seconds, the grade is 40.
2. If the computing time of your program is between X seconds and Y seconds, the grade is $\left\lceil\left(\frac{X-your\ computing\ time}{X-Y}\right)*60\right\rceil+40$.
3. If the computing time of your program is less than Y seconds, the grade is $\mathrm{Y}-\text{your computing time}+100$.
4. If your codes can’t be compiled on the test computer, the grade is 0.

BitmapPlusPlus.hpp

```
#pragma once

#include <fstream>        // std::*fstream
#include <vector>          // std::vector
#include <memory>          // std::unique_ptr
#include <algorithm>    // std::fill
#include <cstdint>        // std::int*_t
#include <cstddef>        // std::size_t
#include <string>          // std::string
#include <cstring>        // std::memcmp
#include <filesystem> // std::filesystem::path
#include <stdexcept>    // std::runtime_error
#include <utility>        // std::exchange

namespace bmp {
   // Magic number for Bitmap .bmp 24 bpp files (24/8 = 3 = rgb colors only)
   static constexpr std::uint16_t BITMAP_BUFFER_MAGIC = 0x4D42;

#pragma pack(push, 1)
   struct BitmapHeader {
      /* Bitmap file header structure */
      std::uint16_t magic;              /* Magic number for file always BM which is 
0x4D42 */
      std::uint32_t file_size;      /* Size of file */
      std::uint16_t reserved1;      /* Reserved */
      std::uint16_t reserved2;      /* Reserved */
      std::uint32_t offset_bits; /* Offset to bitmap data */
      /* Bitmap file info structure */
      std::uint32_t size;                          /* Size of info header */
      std::int32_t width;                           /* Width of image */
      std::int32_t height;                         /* Height of image */
      std::uint16_t planes;                      /* Number of color planes */
      std::uint16_t bits_per_pixel;        /* Number of bits per pixel */
      std::uint32_t compression;              /* Type of compression to use */
      std::uint32_t size_image;               /* Size of image data */
      std::int32_t x_pixels_per_meter; /* X pixels per meter */
```

```
        std::int32_t y_pixels_per_meter; /* Y pixels per meter */
        std::uint32_t clr_used;                     /* Number of colors used */
        std::uint32_t clr_important;          /* Number of important colors */
    };
    // Sanity check
    static_assert(sizeof(BitmapHeader) == 54, "Bitmap header size must be 54 bytes");

    struct Pixel {
        std::uint8_t r; /* Blue value */
        std::uint8_t g; /* Green value */
        std::uint8_t b; /* Red value */

        constexpr Pixel() noexcept : r(0), g(0), b(0) {
        }

        explicit constexpr Pixel(const std::int32_t rgb) noexcept : r((rgb >> 16) & 0xff),
g((rgb >> 8) & 0xff), b((rgb >> 0x0) & 0xff) {
        }

        constexpr Pixel(const std::uint8_t red, const std::uint8_t green, const
std::uint8_t blue) noexcept : r(red), g(green), b(blue) {
        }

        constexpr bool operator==(const Pixel &other) const noexcept {
            if (this == std::addressof(other))
                return true;
            return r == other.r && g == other.g && b == other.b;
        }

        constexpr bool operator!=(const Pixel &other) const noexcept { return !((*this)
== other); }
    };

    static_assert(sizeof(Pixel) == 3, "Bitmap Pixel size must be 3 bytes");
#pragma pack(pop)

    static constexpr Pixel Aqua{0, 255, 255};
    static constexpr Pixel Beige{245, 245, 220};
    static constexpr Pixel Black{0, 0, 0};
    static constexpr Pixel Blue{0, 0, 255};
    static constexpr Pixel Brown{165, 42, 42};
    static constexpr Pixel Chocolate{210, 105, 30};
    static constexpr Pixel Coral{255, 127, 80};
    static constexpr Pixel Crimson{220, 20, 60};
    static constexpr Pixel Cyan{0, 255, 255};
    static constexpr Pixel Firebrick{178, 34, 34};
    static constexpr Pixel Gold{255, 215, 0};
    static constexpr Pixel Gray{128, 128, 128};
    static constexpr Pixel Green{0, 255, 0};
    static constexpr Pixel Indigo{75, 0, 130};
    static constexpr Pixel Lavender{230, 230, 250};
    static constexpr Pixel Lime{0, 255, 0};
    static constexpr Pixel Magenta{255, 0, 255};
    static constexpr Pixel Maroon{128, 0, 0};
    static constexpr Pixel Navy{0, 0, 128};
    static constexpr Pixel Olive{128, 128, 0};
    static constexpr Pixel Orange{255, 165, 0};
    static constexpr Pixel Pink{255, 192, 203};
    static constexpr Pixel Purple{128, 0, 128};
    static constexpr Pixel Red{255, 0, 0};
    static constexpr Pixel Salmon{250, 128, 114};
    static constexpr Pixel Silver{192, 192, 192};
    static constexpr Pixel Snow{255, 250, 250};
    static constexpr Pixel Teal{0, 128, 128};
    static constexpr Pixel Tomato{255, 99, 71};
    static constexpr Pixel Turquoise{64, 224, 208};
    static constexpr Pixel Violet{238, 130, 238};
    static constexpr Pixel White{255, 255, 255};
    static constexpr Pixel Wheat{245, 222, 179};
    static constexpr Pixel Yellow{255, 255, 0};

    class Exception : public std::runtime_error {
    public:
        explicit Exception(const std::string &message) : std::runtime_error(message) {
        }
    };
```

```
    class Bitmap {
    public:
        Bitmap() noexcept : m_pixels(), m_width(0), m_height(0) {
        }

        explicit Bitmap(const std::string &filename) : m_pixels(), m_width(0),
m_height(0) {
            this->load(filename);
        }

        Bitmap(const std::int32_t width, const std::int32_t height)
            : m_pixels(static_cast<std::size_t>(width) * static_cast<std::size_t>(height)),
                m_width(width),
                m_height(height) {
            if (width == 0 || height == 0)
                throw Exception("Bitmap width and height must be > 0");
        }

        Bitmap(const Bitmap &other) = default; // Copy Constructor

        Bitmap(Bitmap &&other) noexcept
            : m_pixels(std::move(other.m_pixels)),
                m_width(std::exchange(other.m_width, 0)),
                m_height(std::exchange(other.m_height, 0)) {
        }

        virtual ~Bitmap() noexcept = default;

    public: /* Draw Primitives */
        /**
         * Draw a line form (x1, y1) to (x2, y2)
         */
        void draw_line(std::int32_t x1, std::int32_t y1, std::int32_t x2, std::int32_t y2,
const Pixel color) {
            const std::int32_t dx = std::abs(x2 - x1);
            const std::int32_t dy = std::abs(y2 - y1);
            const std::int32_t sx = (x1 < x2) ? 1 : -1;
            const std::int32_t sy = (y1 < y2) ? 1 : -1;
            std::int32_t err = dx - dy;
            while (true) {
                m_pixels[IX(x1, y1)] = color;

                if (x1 == x2 && y1 == y2) {
                    break;
                }

                int e2 = 2 * err;
                if (e2 > -dy) {
                    err -= dy;
                    x1 += sx;
                }
                if (e2 < dx) {
                    err += dx;
                    y1 += sy;
                }
            }
        }

        /**
          * Draw a filled rect
          */
        void fill_rect(const std::int32_t x, const std::int32_t y, const std::int32_t width,
const std::int32_t height,
                                  const Pixel color) {
            if (!in_bounds(x, y) || !in_bounds(x + (width - 1), y + (height - 1)))
                throw Exception(
                    "Bitmap::fill_rect(" + std::to_string(x) + ", " + std::to_string(y) + ", " +
std::to_string(width) + ", " +
                    std::to_string(height) + "): x,y,w or h out of bounds");

            for (std::int32_t dx = x; dx < x + width; ++dx) {
                for (std::int32_t dy = y; dy < y + height; ++dy) {
                    m_pixels[IX(dx, dy)] = color;
                }
            }
```

```
    }

    /**
     * Draw a rect (not filled, border only)
     */
    void draw_rect(const std::int32_t x, const std::int32_t y, const std::int32_t
width, const std::int32_t height,
                   const Pixel color) {
        if (!in_bounds(x, y) || !in_bounds(x + (width - 1), y + (height - 1)))
            throw Exception(
                "Bitmap::draw_rect(" + std::to_string(x) + ", " + std::to_string(y) + ", " +
std::to_string(width) + ", " +
                std::to_string(height) + "): x,y,w or h out of bounds");

        for (std::int32_t dx = x; dx < x + width; ++dx) {
            m_pixels[IX(dx, y)] = color;                  // top
            m_pixels[IX(dx, y + height - 1)] = color; // bottom
        }
        for (std::int32_t dy = y; dy < y + height; ++dy) {
            m_pixels[IX(x, dy)] = color;                 // left
            m_pixels[IX(x + width - 1, dy)] = color; // right
        }
    }


    /**
     * Draw a triangle (not filled, border only)
     */
    void draw_triangle(const std::int32_t x1, const std::int32_t y1,
                       const std::int32_t x2, const std::int32_t y2,
                       const std::int32_t x3, const std::int32_t y3,
                       const Pixel color) {
        if (!in_bounds(x1, y1) || !in_bounds(x2, y2) || !in_bounds(x3, y3))
            throw Exception("Bitmap::draw_triangle: One or more points are out of
bounds");

        draw_line(x1, y1, x2, y2, color);
        draw_line(x2, y2, x3, y3, color);
        draw_line(x3, y3, x1, y1, color);
    }

    /**
     * Draw a filled triangle
     */
    void fill_triangle(const std::int32_t x1, const std::int32_t y1,
                       const std::int32_t x2, const std::int32_t y2,
                       const std::int32_t x3, const std::int32_t y3,
                       const Pixel color) {
        if (!in_bounds(x1, y1) || !in_bounds(x2, y2) || !in_bounds(x3, y3))
            throw Exception("Bitmap::fill_triangle: One or more points are out of
bounds");

        // Sort the vertices by y-coordinate (top to bottom)
        std::vector<std::pair<std::int32_t, std::int32_t> > vertices = {
            {x1, y1},
            {x2, y2},
            {x3, y3}};
        std::sort(vertices.begin(), vertices.end(), [](const auto &a, const auto &b) {
            return a.second < b.second;
        });

        const auto [x_top, y_top] = vertices[0];
        const auto [x_mid, y_mid] = vertices[1];
        const auto [x_bot, y_bot] = vertices[2];

        // Calculate the slopes of the left and right edges
        const float slope_left = static_cast<float>(x_mid - x_top) / (y_mid - y_top);
        const float slope_right = static_cast<float>(x_bot - x_top) / (y_bot - y_top);

        // Initialize the starting and ending x-coordinates for each scanline
        std::vector<std::pair<std::int32_t, std::int32_t> > scanlines(y_mid - y_top + 1);
        for (std::int32_t y = y_top; y <= y_mid; ++y) {
            const auto x_start = static_cast<std::int32_t>(x_top + (y - y_top) *
slope_left);
            const auto x_end = static_cast<std::int32_t>(x_top + (y - y_top) *
slope_right);
```

```
            scanlines[y - y_top] = {x_start, x_end};
        }

        // Fill the upper part of the triangle
        for (std::int32_t y = y_top; y <= y_mid; ++y) {
            const std::int32_t x_start = scanlines[y - y_top].first;
            const std::int32_t x_end = scanlines[y - y_top].second;
            draw_line(x_start, y, x_end, y, color);
        }

        // Update the slope for the right edge of the triangle
        const float new_slope_right = static_cast<float>(x_bot - x_mid) / (y_bot -
y_mid);

        // Update the x-coordinates for the scanlines in the lower part of the triangle
        for (std::int32_t y = y_mid + 1; y <= y_bot; ++y) {
            const auto x_start = static_cast<std::int32_t>(x_mid + (y - y_mid) *
slope_left);
            const auto x_end = static_cast<std::int32_t>(x_top + (y - y_top) *
new_slope_right);
            scanlines[y - y_top] = {x_start, x_end};
        }

        // Fill the lower part of the triangle
        for (std::int32_t y = y_mid + 1; y <= y_bot; ++y) {
            const std::int32_t x_start = scanlines[y - y_top].first;
            const std::int32_t x_end = scanlines[y - y_top].second;
            draw_line(x_start, y, x_end, y, color);
        }
    }

    /**
     * Draw a circle with a given center and radius
     */
    void draw_circle(const std::int32_t center_x, const std::int32_t center_y, const
std::int32_t radius,
                              const Pixel color) {
```

```
        if (!in_bounds(center_x - radius, center_y - radius) || !in_bounds(center_x +
radius, center_y + radius))
            throw Exception("Bitmap::draw_circle: Circle exceeds bounds");

        std::int32_t x = radius;
        std::int32_t y = 0;
        std::int32_t err = 0;

        while (x >= y) {
            // Draw pixels in all octants
            m_pixels[IX(center_x + x, center_y + y)] = color;
            m_pixels[IX(center_x + y, center_y + x)] = color;
            m_pixels[IX(center_x - y, center_y + x)] = color;
            m_pixels[IX(center_x - x, center_y + y)] = color;
            m_pixels[IX(center_x - x, center_y - y)] = color;
            m_pixels[IX(center_x - y, center_y - x)] = color;
            m_pixels[IX(center_x + y, center_y - x)] = color;
            m_pixels[IX(center_x + x, center_y - y)] = color;

            // Update error and y for the next pixel
            if (err <= 0) {
                y += 1;
                err += 2 * y + 1;
            }

            // Update error and x for the next pixel
            if (err > 0) {
                x -= 1;
                err -= 2 * x + 1;
            }
        }
    }

    /**
     * Fill a circle with a given center and radius
     */
    void fill_circle(const std::int32_t center_x, const std::int32_t center_y, const
std::int32_t radius,
                                const Pixel color) {
        if (!in_bounds(center_x - radius, center_y - radius) || !in_bounds(center_x +
radius, center_y + radius))
            throw Exception("Bitmap::fill_circle: Circle exceeds bounds");

        std::int32_t x = radius;
        std::int32_t y = 0;
        std::int32_t err = 0;

        while (x >= y) {
            // Fill scanlines in all octants
            for (std::int32_t i = center_x - x; i <= center_x + x; ++i) {
                m_pixels[IX(i, center_y + y)] = color;
                m_pixels[IX(i, center_y - y)] = color;
            }

            for (std::int32_t i = center_x - y; i <= center_x + y; ++i) {
                m_pixels[IX(i, center_y + x)] = color;
                m_pixels[IX(i, center_y - x)] = color;
            }

            // Update error and y for the next pixel
            if (err <= 0) {
                y += 1;
                err += 2 * y + 1;
            }

            // Update error and x for the next pixel
            if (err > 0) {
                x -= 1;
                err -= 2 * x + 1;
            }
        }
    }

public: /* Accessors */
    /**
     *    Get pixel at position x,y
```

```
     */
    Pixel &get(const std::int32_t x, const std::int32_t y) {
        if (!in_bounds(x, y))
            throw Exception("Bitmap::get(" + std::to_string(x) + ", " + std::to_string(y) +
"): x,y out of bounds");
        return m_pixels[IX(x, y)];
    }

    /**
     *    Get const pixel at position x,y
     */
    [[nodiscard]] const Pixel &get(const std::int32_t x, const std::int32_t y) const {
        if (!in_bounds(x, y))
            throw Exception("Bitmap::get(" + std::to_string(x) + ", " + std::to_string(y) +
"): x,y out of bounds");
        return m_pixels[IX(x, y)];
    }

    /**
     *    Returns the width of the Bitmap image
     */
    [[nodiscard]] std::int32_t width() const noexcept { return m_width; }

    /**
     *    Returns the height of the Bitmap image
     */
    [[nodiscard]] std::int32_t height() const noexcept { return m_height; }

    /**
     *    Clears Bitmap pixels with an rgb color
     */
    void clear(const Pixel pixel = Black) {
        std::fill(m_pixels.begin(), m_pixels.end(), pixel);
    }

  public: /* Operators */
    const Pixel &operator[](const std::size_t i) const { return m_pixels[i]; }
```

```
        Pixel &operator[](const std::size_t i) { return m_pixels[i]; }

        bool operator!() const noexcept { return (m_pixels.empty()) || (m_width == 0)
|| (m_height == 0); }

        explicit operator bool() const noexcept { return !(*this); }

        bool operator==(const Bitmap &image) const {
            if (this == std::addressof(image)) {
                return true;
            }
            return (m_width == image.m_width) &&
                        (m_height == image.m_height) &&
                        (std::memcmp(m_pixels.data(), image.m_pixels.data(), sizeof(Pixel) *
m_pixels.size()) == 0);
        }

        bool operator!=(const Bitmap &image) const { return !(*this == image); }

        Bitmap &operator=(const Bitmap &image) // Copy assignment operator
        {
            if (this != std::addressof(image)) {
                m_width = image.m_width;
                m_height = image.m_height;
                m_pixels = image.m_pixels;
            }
            return *this;
        }

        Bitmap &operator=(Bitmap &&image) noexcept {
            if (this != std::addressof(image)) {
                m_pixels = std::move(image.m_pixels);
                m_width = std::exchange(image.m_width, 0);
                m_height = std::exchange(image.m_height, 0);
            }
            return *this;
        }
```

```
    public: /** foreach iterators access */
        [[nodiscard]] std::vector<Pixel>::iterator begin() noexcept { return
m_pixels.begin(); }

        [[nodiscard]] std::vector<Pixel>::iterator end() noexcept { return
m_pixels.end(); }

        [[nodiscard]] std::vector<Pixel>::const_iterator cbegin() const noexcept { return
m_pixels.cbegin(); }

        [[nodiscard]] std::vector<Pixel>::const_iterator cend() const noexcept { return
m_pixels.cend(); }

        [[nodiscard]] std::vector<Pixel>::reverse_iterator rbegin() noexcept { return
m_pixels.rbegin(); }

        [[nodiscard]] std::vector<Pixel>::reverse_iterator rend() noexcept { return
m_pixels.rend(); }

        [[nodiscard]] std::vector<Pixel>::const_reverse_iterator crbegin() const noexcept
{ return m_pixels.crbegin(); }

        [[nodiscard]] std::vector<Pixel>::const_reverse_iterator crend() const noexcept
{ return m_pixels.crend(); }

    public: /* Modifiers */
        /**
          *    Sets rgb color to pixel at position x,y
          *      @throws bmp::Exception on error
          */
        void set(const std::int32_t x, const std::int32_t y, const Pixel color) {
            if (!in_bounds(x, y)) {
                throw Exception("Bitmap::set(" + std::to_string(x) + ", " + std::to_string(y) +
"): x,y out of bounds");
            }
            m_pixels[IX(x, y)] = color;
        }
```

```
/**
*      Vertically flips the bitmap and returns the flipped version
*
*/
[[nodiscard("Bitmap::flip_v() is immutable")]]
Bitmap flip_v() const {
    Bitmap finished(m_width, m_height);
    for (std::int32_t x = 0; x < m_width; ++x) {
        for (std::int32_t y = 0; y < m_height; ++y) {
            // Calculate the reverse y-index
            finished.m_pixels[IX(x, y)] = m_pixels[IX(x, m_height - 1 - y)];
        }
    }
    return finished;
}

/**
*      Horizontally flips the bitmap and returns the flipped version
*
*/
[[nodiscard("Bitmap::flip_h() is immutable")]]
Bitmap flip_h() const {
    Bitmap finished(m_width, m_height);
    for (std::int32_t y = 0; y < m_height; ++y) {
        for (std::int32_t x = 0; x < m_width; ++x) {
            // Calculate the reverse x-index
            finished.m_pixels[IX(x, y)] = m_pixels[IX(m_width - 1 - x, y)];
        }
    }
    return finished;
}

/**
*      Rotates the bitmap to the right and returns the rotated version
*
*/
[[nodiscard("Bitmap::rotate_90_left() is immutable")]]
```

```
    Bitmap rotate_90_left() const {
        Bitmap finished(m_height, m_width); // Swap dimensions

        for (std::int32_t y = 0; y < m_height; ++y) {
            const std::int32_t y_offset = y * m_width; // Precompute row start index
            for (std::int32_t x = 0; x < m_width; ++x) {
                // Original pixel at (x, y) moves to (y, m_width - 1 - x)
                finished.m_pixels[(m_width - 1 - x) * m_height + y] = m_pixels[y_offset +
x];
            }
        }

        return finished;
    }

    /**
    *     Rotates the bitmap to the left and returns the rotated version
    *
    */
    [[nodiscard("Bitmap::rotate_90_right() is immutable")]]
    Bitmap rotate_90_right() const {
        Bitmap finished(m_height, m_width); // Swap dimensions
        for (std::int32_t y = 0; y < m_height; ++y) {
            const std::int32_t y_offset = y * m_width; // Precompute row start index
            for (std::int32_t x = 0; x < m_width; ++x) {
                finished.m_pixels[x * m_height + (m_height - 1 - y)] = m_pixels[y_offset
+ x];
            }
        }

        return finished;
    }

    /**
     *   Saves Bitmap pixels into a file
     *     @throws bmp::Exception on error
     */
    void save(const std::filesystem::path &filename) const {
        // Calculate row and bitmap size
        const std::int32_t row_size = m_width * 3 + m_width % 4;
        const std::uint32_t bitmap_size = row_size * m_height;

        // Construct bitmap header
        BitmapHeader header{};
        /* Bitmap file header structure */
        header.magic = BITMAP_BUFFER_MAGIC;
        header.file_size = bitmap_size + sizeof(BitmapHeader);
        header.reserved1 = 0;
        header.reserved2 = 0;
        header.offset_bits = sizeof(BitmapHeader);
        /* Bitmap file info structure */
        header.size = 40;
        header.width = m_width;
        header.height = m_height;
        header.planes = 1;
        header.bits_per_pixel = sizeof(Pixel) * 8; // 24bpp
        header.compression = 0;
        header.size_image = bitmap_size;
        header.x_pixels_per_meter = 0;
        header.y_pixels_per_meter = 0;
        header.clr_used = 0;
        header.clr_important = 0;

        // Save bitmap to output file
        if (std::ofstream ofs{filename, std::ios::binary}; ofs.good()) {
            // Write Header
            ofs.write(reinterpret_cast<const char *>(&header), sizeof(BitmapHeader));
            if (!ofs.good()) {
                throw Exception("Bitmap::save(\"" + filename.string() + "\"): Failed to
write bitmap header to file.");
            }

            // Write Pixels
            std::vector<std::uint8_t> line(row_size);
            for (std::int32_t y = m_height - 1; y >= 0; --y) {
                std::size_t i = 0;
                for (std::int32_t x = 0; x < m_width; ++x) {
                    const Pixel &color = m_pixels[IX(x, y)];
                    line[i++] = color.b;
                    line[i++] = color.g;
                    line[i++] = color.r;
                }
                ofs.write(reinterpret_cast<const char *>(line.data()), line.size());
                if (!ofs.good()) {
                    throw Exception("Bitmap::save(\"" + filename.string() + "\"): Failed to
write bitmap pixels to file.");
                }
            }
        } else
            throw Exception("Bitmap::save(\"" + filename.string() + "\"): Failed to open
file.");
    }

    /**
     *   Loads Bitmap from file
     *     @throws bmp::Exception on error
     */
    void load(const std::filesystem::path &filename) {
        m_pixels.clear();

        if (std::ifstream ifs{filename, std::ios::binary}; ifs.good()) {
            // Read Header
            std::unique_ptr<BitmapHeader> header(new BitmapHeader());
            ifs.read(reinterpret_cast<char *>(header.get()), sizeof(BitmapHeader));

            // Check if Bitmap file is valid
            if (header->magic != BITMAP_BUFFER_MAGIC) {
                throw Exception("Bitmap::load(\"" + filename.string() + "\"):
Unrecognized file format.");
            }
            // Check if the Bitmap file has 24 bits per pixel (for now supporting only
24bpp bitmaps)
            if (header->bits_per_pixel != 24) {
                throw Exception("Bitmap::load(\"" + filename.string() + "\"): Only 24 bits
per pixel bitmaps supported.");
            }

            // Seek the beginning of the pixels data
            // Note: We can't just assume we're there right after we read the
BitmapHeader
            // Because some editors like Gimp might put extra information after the
header.
            // Thanks to @seeliger-ec
            ifs.seekg(header->offset_bits);

            // Set width & height
            m_width = header->width;
            m_height = header->height;

            // Resize pixels size
            m_pixels.resize(static_cast<std::size_t>(m_width) *
static_cast<std::size_t>(m_height), Black);

            // Read Bitmap pixels
            const std::int32_t row_size = m_width * 3 + m_width % 4;
            std::vector<std::uint8_t> line(row_size);
            for (std::int32_t y = m_height - 1; y >= 0; --y) {
                ifs.read(reinterpret_cast<char *>(line.data()), line.size());
                if (!ifs.good())
                    throw Exception("Bitmap::load(\"" + filename.string() + "\"): Failed to
read bitmap pixels from file.");
                std::size_t i = 0;
                for (std::int32_t x = 0; x < m_width; ++x) {
                    Pixel color{};
                    color.b = line[i++];
                    color.g = line[i++];
                    color.r = line[i++];
                    m_pixels[IX(x, y)] = color;
                }
            }
        } else
```

```
                throw Exception("Bitmap::load(\"" + filename.string() + "\"): Failed to load
bitmap pixels from file.");
        }

    private: /* Utils */
        /**
          *    Converts 2D x,y coords into 1D index
          */
        [[nodiscard]] constexpr std::size_t IX(const std::int32_t x, const std::int32_t y)
const noexcept {
            return static_cast<std::size_t>(x) + static_cast<std::size_t>(m_width) *
static_cast<std::size_t>(y);
        }
        /**
          *    Returns true if x,y coords are within boundaries
          */
        [[nodiscard]] constexpr bool in_bounds(const std::int32_t x, const std::int32_t y)
const noexcept {
            return (x >= 0) && (x < m_width) && (y >= 0) && (y < m_height);
        }

    private:
        std::vector<Pixel> m_pixels;
        std::int32_t m_width;
        std::int32_t m_height;
    };
}
```

single thread sobel.cpp

```
#include <filesystem>
#include <iostream>
#include <ctime>
#include <cmath>

#include "BitmapPlusPlus.hpp"

inline uint8_t boundPixel(double val) {
	if (val < 0)
		return 0;
	if (val > 255)
		return 255;
	return static_cast<uint8_t>(round(val));
}

void filtingRow(bmp::Bitmap &des, const bmp::Bitmap &src, const int row, const
double W[], const int kSize) {
	const int offset = kSize >> 1;
	const int len = kSize * kSize;
	std::vector<bmp::Pixel>::const_iterator *A = new
std::vector<bmp::Pixel>::const_iterator [len];
	for (int i = 0, r = -offset; r <= offset; ++r) {
		A[i++] = src.cbegin() + (row + r) * src.width();
		for (int j = -offset + 1; j <= offset; ++j, ++i)
			A[i] = A[i-1] + 1;
	}

	std::vector<bmp::Pixel>::iterator tar = des.begin() + row * des.width() + offset;
	for (int i = kSize - 1; i < des.width(); ++i) {
		double r = 0, g = 0, b = 0;
		for (int j = 0; j < len; ++j) {
			r += W[j] * A[j]->r;
			g += W[j] * A[j]->g;
			b += W[j] * A[j]->b;
			++A[j];
		}
```

```
                *tar = bmp::Pixel(boundPixel(r), boundPixel(g), boundPixel(b));
                ++tar;

        }
        delete [] A;
}

int main(int argc, char** argv) {
        if (argc < 4) {
                std::cout << "Usage: " << argv[0] << " <SOURCE_BMP> <TARGET_BMP>
<AMOUNT OF FILE>";
                return 0;
        }
        try {
                char infilename[25],outfilename[25];
                int amountoffile=atoi(argv[3]);
                auto beginTime = std::chrono::steady_clock::now();
                const double W[] = {
                        1, 0, -1,
                        2, 0, -2,
                        1, 0, -1
                };
                for (int filecount=0;filecount<amountoffile;filecount++)
                        {
                        sprintf(infilename,"%s%d.bmp",argv[1],filecount);
                        sprintf(outfilename,"%s%d.bmp",argv[2],filecount);
                        bmp::Bitmap srcImage;
                        srcImage.load(infilename);
                        bmp::Bitmap desImage(srcImage.width(), srcImage.height());
                        for (int R = 1; R < srcImage.height() - 1; ++R)
                                filtingRow(desImage, srcImage, R, W, 3);
                        desImage.save(outfilename);
                        }
                auto endTime = std::chrono::steady_clock::now();
                double totaltime = std::chrono::duration<double>(endTime -
beginTime).count();
                std::cout << "Takes "<<totaltime<< "secs" << std::endl;
        } catch (const bmp::Exception &e) {
```

```
		return EXIT_FAILURE;
	}
	return 0;
}
```