CS219 Project 4 Report

Zhige Chen 12413315 SUSTech

2025 Spring Semester

Contents

1 Overview

MGIL (Modern Generic Image Processing Library) is a modern C++ image processing library that reinvents generic image manipulation paradigms by combining the proven architecture of Boost.GIL with cutting-edge modern C++ feature. Designed for high composability, extensibility, and performance, MGIL introduces a separation between view-based lazy evaluation and eager algorithm execution while providing STL-compatible interfaces and user-friendly pipe operator syntax.

2 Features

- Single header library
- User-friendly pipe operator syntax
- STL-compatible and intuitive interfaces
- Highly extensible and customizable
- Verifies itself upon include

3 Build Instructions

MGIL is built upon newest C++ features, including the features that haven't passed the whole C++ standardization process or have less compiler support, so it also requires latest compilers to compile. So far MGIL has only be built and fully tested on **Clang 21** (built from latest source from the LLVM git repository) with **libc++ 20.1.3** and being partially tested on Clang 20.1.3 with libc++ 20.1.3. **MSVC** is completely unsupported due to its poor support for the latest C++ features, GCC (14 or newer) is also unsupported due to missing of several critical C++ standard library features (e.g. `std::mdspan`).

4 Example Use Cases

4.1 A quick tour of the interfaces

```cpp
#include "mgil.hpp"

#include <print>

auto main() -> int {
    using namespace mgil;

    using pixel = Pixel<int, rgb_layout_t>;

    // Generate an image view of a gradient image. width = 8, height = 8,
    ↪  initial={0, 0, 0}, x step={1, 2, 0}, y
    // step={0, 0, 3}
    auto gradient_view = gradient(8, 8,
        pixel{0, 0, 0}, pixel{1, 2, 0}, pixel{0, 0, 3});
    // STL compatibility: just print the image view!
```

```
    // Both the std::cout and the newer std::println() are supported.
    // You can also use the ranged-for or old-fashioned iterator loop to manually
    ↪  do the printing
    std::println("{}", gradient_view);

    // Let play with the image view!
    // The MGIL view adaptors can be chained together using the pipe operator "|"
    // Just like the C++ Ranges library!
    // Read adaptor1 | adaptor 2 as adaptor2(adaptor1)
    auto processed_view = gradient_view
            | crop(2, 2, 4, 4)  // Crop the view at (2, 2) with cropped view
            ↪  width = 4, height = 4
            | rotate90()        // Rotate the view by 90 degrees clockwise
            | flipVertical()    // Flip the view vertically
            | transform([](pixel input) {
                    input.get<red_color_t>() = 0;     // Make red channel all
                    ↪  zero
                    input.get<green_color_t>() *= 2;  // Amplify the green
                    ↪  channel by factor of 2
                    return input;
                });

    // Print out the view to see what happened!
    std::println("{}", processed_view);
}
```

Program output:

```
ImageView(width = 8, height = 8):
{0,0,0} {1,2,0} {2,4,0} {3,6,0} {4,8,0} {5,10,0} {6,12,0} {7,14,0}
{0,0,3} {1,2,3} {2,4,3} {3,6,3} {4,8,3} {5,10,3} {6,12,3} {7,14,3}
{0,0,6} {1,2,6} {2,4,6} {3,6,6} {4,8,6} {5,10,6} {6,12,6} {7,14,6}
{0,0,9} {1,2,9} {2,4,9} {3,6,9} {4,8,9} {5,10,9} {6,12,9} {7,14,9}
{0,0,12} {1,2,12} {2,4,12} {3,6,12} {4,8,12} {5,10,12} {6,12,12} {7,14,12}
{0,0,15} {1,2,15} {2,4,15} {3,6,15} {4,8,15} {5,10,15} {6,12,15} {7,14,15}
{0,0,18} {1,2,18} {2,4,18} {3,6,18} {4,8,18} {5,10,18} {6,12,18} {7,14,18}
{0,0,21} {1,2,21} {2,4,21} {3,6,21} {4,8,21} {5,10,21} {6,12,21} {7,14,21}

ImageView(width = 4, height = 4):
{0,20,15} {0,20,12} {0,20,9} {0,20,6}
{0,16,15} {0,16,12} {0,16,9} {0,16,6}
{0,12,15} {0,12,12} {0,12,9} {0,12,6}
{0,8,15} {0,8,12} {0,8,9} {0,8,6}
```

4.2 STL compatibility: Structured binding, STL algorithms, and C++ ranges

```
#include "mgil.hpp"

#include <print>
#include <vector>
#include <algorithm>

auto main() -> int {
    using namespace mgil;

    // MGIL points and pixels all implemented the C++ tuple protocol, so you can
    // retrieve their components in an intuitive way, by using the structured
    // binding

    using pixel = Pixel<int, rgb_layout_t>;

    pixel p{11, 45, 14};
    auto [r, g, b] = p;
```

```
    std::println("R: {}, G: {}, B: {}", r, g, b);

    using point = Point<std::ptrdiff_t>;
    point pos{42, 24};
    auto [x, y] = pos;
    std::println("x: {}, y: {}", x, y);

    // You can create image views from C++ ranges easily:
    using gray_pixel = Pixel<int, gray_layout_t>;
    // from either a one-dimensional range with width and height specified...
    std::vector pixels1 = {
        gray_pixel{1}, gray_pixel{2}, gray_pixel{3},
        gray_pixel{4}, gray_pixel{5}, gray_pixel{6},
        gray_pixel{7}, gray_pixel{8}, gray_pixel{9},
    };
    auto from_range_view_1 = fromRange(pixels1, 3, 3);
    std::println("from_range_view_1: {}", from_range_view_1);
    // or a two-dimensional nested range
    std::vector pixels2 = {
        std::vector{gray_pixel{1}, gray_pixel{2}, gray_pixel{3}},
        std::vector{gray_pixel{4}, gray_pixel{5}, gray_pixel{6}},
        std::vector{gray_pixel{7}, gray_pixel{8}, gray_pixel{9}},
    };
    auto from_range_view_2 = fromRange(pixels2);
    std::println("from_range_view_2: {}", from_range_view_2);

    // MGIL views can integrate seamlessly with STL algorithms:
    auto max_pixel = std::ranges::max_element(from_range_view_1);
    std::println("max_pixel: {}", *max_pixel);
}
```

Program output:

```
R: 11, G: 45, B: 14
x: 42, y: 24
from_range_view_1: ImageView(width = 3, height = 3):
{1} {2} {3}
{4} {5} {6}
{7} {8} {9}

from_range_view_2: ImageView(width = 3, height = 3):
{1} {2} {3}
{4} {5} {6}
{7} {8} {9}

max_pixel: {9}
```

4.3 A more complex example with image file I/O

```
#include "mgil.hpp"

auto main() -> int {
    using namespace mgil;

    // You can read image files with a unified interface readImage().
    // The support for different image format is provided by different
    // image file I/O class.
    auto image = readImage<BMPFileIO<>>("./demo3.bmp");

    // Handle possible exceptions using C++23 std::expected
    if (not image.has_value()) {
        // The error code is provided by an enum class
        std::println("Failed to read image with error code {}",
            std::to_underlying(image.error()));
```

```
        return 1;
    }

    // Now we can play with the image!
    // The readImage will return an owning image container.
    // To use it with the view adaptors, first you should convert it into a view
    // by the .toView() function
    auto width = image.value().toView().width() / 2;
    auto height = image.value().toView().height() / 2;
    auto processed_view_1 = image.value().toView()
        | rotate180()   // rotate the image by 180 degrees clockwise
        | transform([](auto pixel) {
            pixel.template get<red_color_t>() = 0;
            pixel.template get<green_color_t>() = 0;
            // extracts the blue channel by setting other two all zero
            return pixel;
        })
        | nearest(width, height);   // subsample the image by nearest linear
        ↪  interpolation

    auto processed_view_2 = image.value().toView()
        | flipVertical()    // flip the image vertically
        | transform([](auto pixel) {
            pixel.template get<red_color_t>() = 0;
            pixel.template get<blue_color_t>() = 0;
            // extracts the green channel by setting other two all zero
            return pixel;
        })
        | nearest(width, height);   // subsample the image by nearest linear
        ↪  interpolation

    // concat the two image views vertically
    auto final_processed_view = concatVertical(processed_view_1,
    ↪  processed_view_2);

    // then we can write the view to file to see what happened!
    writeImage<BMPFileIO<>>(final_processed_view, "./demo3_result.bmp");
}
```

Program output:

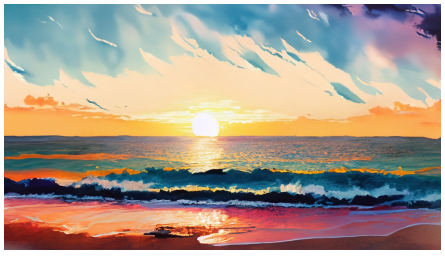

Figure 1: Input Image

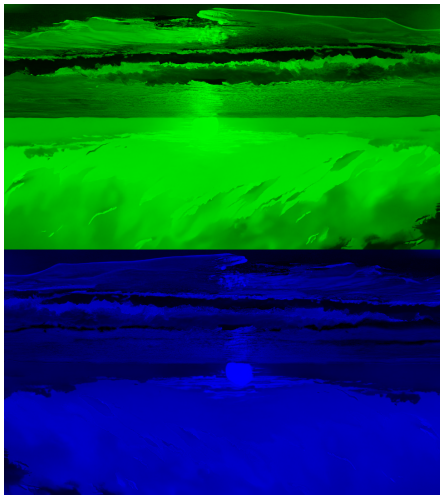

Figure 2: Result Image

4.4 Algorithms: Gaussian blur

```cpp
#include "mgil.h"

auto main() -> int {
    using namespace mgil;

    auto image = readImage<BMPFileIO<>>("./demo4.bmp");

    // Handle possible exceptions using C++23 std::expected
    if (not image.has_value()) {
        // The error code is provided by an enum class
        std::println("Failed to read image with error code {}",
            std::to_underlying(image.error()));
        return 1;
    }

    // Using the Gaussian blur algorithm to blur the image.
    auto blurred_image = gaussianBlur(image.value().toView(), 10.f);

    // then we can write the view to file
    writeImage<BMPFileIO<>>(blurred_image.toView(), "./demo4_result.bmp");
}
```

Program output:

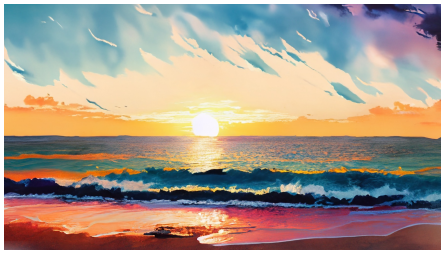

Figure 3: Input Image

Figure 4: Result Image

4.5 User customized image view factory

```cpp
#include "mgil.hpp"

#include <print>

template<typename Pixel, typename View1, typename View2>
    requires mgil::IsPixel<Pixel> and mgil::IsImageView<View1> and
    ↪ mgil::IsImageView<View2> and
            mgil::IsImageViewsCompatible<View1, View2>
class BlendingView {
    struct BlendingDeref : mgil::deref_base<BlendingDeref, Pixel, Pixel &, Pixel
    ↪ const &, Pixel, false> {
        View1 view1;
        View2 view2;
        constexpr auto operator()(mgil::Point<std::ptrdiff_t> const &pos) const
        ↪ -> Pixel {
            // When being dereferenced, return pixel generated by averaging two
            ↪ pixels from the same location in two
            // views
            return view1(pos) / typename View1::value_type::channel_type(2) +
            ↪ view2(pos) / typename View1::value_type::channel_type(2);
        }
    };
    using locator = mgil::position_locator<BlendingDeref>;
```

```
    using point_type = mgil::Point<std::ptrdiff_t>;
    using view_type = mgil::ImageView<locator>;

public:
    constexpr auto operator()(View1 view1, View2 view2) const {
        assert(view1.width() == view2.width());
        assert(view1.height() == view2.height());
        return view_type(view1.width(), view1.height(),
                         locator({0, 0}, {1, 1}, BlendingDeref{.view1 = view1,
                         ↪  .view2 = view2}));
    }
};

template<typename View1, typename View2, typename Pixel = typename
↪  View1::value_type>
    requires mgil::IsPixel<Pixel> and mgil::IsImageView<View1> and
    ↪  mgil::IsImageView<View2> and
            mgil::IsImageViewsCompatible<View1, View2>
constexpr auto blending(View1 view1, View2 view2) {
    return BlendingView<Pixel, View1, View2>{}(view1, view2);
}

auto main() -> int {
    using namespace mgil;

    auto image1 = readImage<BMPFileIO<>>("./demo6_1.bmp");
    auto image2 = readImage<BMPFileIO<>>("./demo6_2.bmp");

    // Handle possible exceptions using C++23 std::expected
    if (not image1.has_value()) {
        // The error code is provided by an enum class
        std::println("Failed to read image with error code {}",
        ↪  std::to_underlying(image1.error()));
        return;
    }
    if (not image2.has_value()) {
        // The error code is provided by an enum class
        std::println("Failed to read image with error code {}",
        ↪  std::to_underlying(image2.error()));
        return;
    }

    // Try our customized image view factory!.
    auto blended_image = blending(image1.value().toView(),
    ↪  image2.value().toView());

    // then we can write the view to file
    writeImage<BMPFileIO<>>(blended_image, "./demo6_result.bmp");
}
```

Program output:

Figure 5: Input Image 1

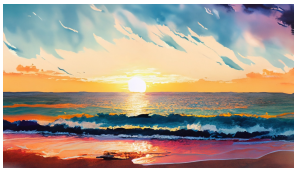

Figure 6: Input Image 2

Figure 7: Result Image

5 Library Design

5.1 Dual Computation Model

Like how the C++ standard library distinguish the lazily-evaluated range adaptors from eagerly-evaluated algorithms. MGIL has **image views** and image algorithms.

Image views are implemented as standard-conforming range factories/adaptors with $O(1)$ construction cost. The computation is deferred until pixel access through iterator interface. And since the views are non-owning, they can be composed without temporary buffers. All these features made image views both fast and memory-efficient. Following is an example demostrating the lazy feature of the views:

```
#include "mgil.h"

auto main() -> int {
    using namespace mgil;

    auto image = readImage<BMPFileIO<>>("./demo4.bmp");

    // Handle possible exceptions using C++23 std::expected
    if (not image.has_value()) {
        // The error code is provided by an enum class
        std::println("Failed to read image with error code {}",
            std::to_underlying(image.error()));
        return 1;
    }

    // Using the Gaussian blur algorithm to blur the image.
    auto blurred_image = gaussianBlur(image.value().toView(), 10.f);

    // then we can write the view to file
    writeImage<BMPFileIO<>>(blurred_image.toView(), "./demo4_result.bmp");
}
```

Program output:

```
generate(0, 0)
transform({0})
Loop: {0}
generate(1, 0)
transform({1})
Loop: {1}
generate(0, 1)
transform({1})
Loop: {1}
generate(1, 1)
transform({2})
Loop: {2}
```

From the output we can see the computation of `generate` and `transform` only happens when a pixel is accessed.

For image algorithms, they guarantee computational completeness upon invocation, this made them have much spatial locality and cache coherence. So the algorithm is more suitable for more heavy tasks, like computing convolution.

5.2 Modern C++ Template Metaprogramming

MGIL fully adopts C++20 concepts to constrain its generic interfaces, this could give much better error messages than traditional SFINAE-based approaches, and make the requirement of interfaces clearer.

Also, by the power of `constexpr`, most of the features of MGIL is usable at compile-time. This can give users more chances of optimizing their programs using MGIL.

6 Possible Enhancements

Due to limited time, MGIL only has limited image processing capability, e.g. only support R/W BMP files, lacking of many more useful algorithms. But thankfully to the structure of the library, extending its capability is very easy.

Also I haven't tweak the performance of the library. By integrating it with an upcoming C++ standard library feature, `std::execution`, MGIL can fully unleash its performance by asynchronous and concurrency.

MGIL currently lacks a comprehensive set of unit tests, which from my view is necessary for a good library.

MGIL relies heavily on extremely complicated template metaprogramming tricks, this could make compile error messages very lengthy and unreadable. With the aid of some upcoming C++ features, like `constexpr` exceptions, the error messages could become more clear and concise.

7 Generative AI Usage

ChatGPT-4o was used to summon a list of common image processing algorithms and their possible implementations. And Deepseek-o1 was used to generate the structure of this project report.

8 MGIL API Reference

8.1 Image View Factories

8.1.1 `mgil::empty`

```
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto empty(std::ptrdiff_t width, std::ptrdiff_t height);
```

Preconditions: `width` ≥ 0, `height` ≥ 0.

Returns: An image view of size `width` $*$ `height` with all its pixels default-constructed.

8.1.2 `mgil::identical`

```
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto identical(std::ptrdiff_t width, std::ptrdiff_t height, Pixel const
↪  &pixel);
```

Preconditions: `width` ≥ 0, `height` ≥ 0.

Returns: An image view of size `width` $*$ `height` with all its pixels equals to `pixel`.

8.1.3 `mgil::gradient`

```
template<typename Pixel>
    requires IsPixel<Pixel>
constexprauto gradient(std::ptrdiff_t width, std::ptrdiff_t height, Pixel const
↪  &start,Pixel const&step_x,Pixel const &step_y);
```

Preconditions: `width` ≥ 0, `height` ≥ 0.

Returns: An image view of size `width` $*$ `height`, where pixel at coordinate (x, y) has value of `start` $+ x *$ `step_x` $+ y *$ `step_y`.

8.1.4 `mgil::generate`

```
template<typename Gen, typename Pixel = std::invoke_result_t<Gen, std::ptrdiff_t,
↪  std::ptrdiff_t>>
    requires IsPixel<Pixel>
constexpr auto generate(std::ptrdiff_t width, std::ptrdiff_t height, Gen
↪  generator);
```

Preconditions: `width` ≥ 0, `height` ≥ 0.

Returns: An image view of size `width` $*$ `height`, where pixel at coordinate (x, y) is generated by the function call `gen(x,y)`.

8.1.5 `mgil::pattern`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto pattern(View view, std::ptrdiff_t width, std::ptrdiff_t height);
```

Preconditions: `width` ≥ 0, `height` ≥ 0.
Returns: An image view of size `width` $*$ `height` generated by repeated the pattern specified by `view`.

8.1.6 `mgil::concatHorizontal`

```
template<typename View1, typename View2, typename Pixel = typename
↪  View1::value_type>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
constexpr auto concatHorizontal(View1 view1, View2 view2);
```

Preconditions: `view1.height()==view2.height()`.
Returns: An image view generated by concatenating two views horizontally.

8.1.7 `mgil::concatVertical`

```
template<typename View1, typename View2, typename Pixel = typename
↪  View1::value_type>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
constexpr auto concatVertical(View1 view1, View2 view2);
```

Preconditions: `view1.width()==view2.width()`.
Returns: An image view generated by concatenating two views vertically.

8.1.8 `mgil::checkerboard`

```
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto checkerboard(std::ptrdiff_t width, std::ptrdiff_t height, Pixel
↪  const &pixel1, Pixel const &pixel2, std::ptrdiff_t cell_width, std::ptrdiff_t
↪  cell_height);
```

Preconditions: `width` ≥ 0, `height` ≥ 0, `cell_width` $\leq$ `width`, `cell_height` $\leq$ `height`.
Returns: A checkerboard of size `width` $*$ `height` with two cell colors specified by `pixel1` and `pixel2`.

8.1.9 `mgil::noiseUniform`

```
template<typename Pixel, typename Gen>
    requires IsPixel<Pixel>
constexpr auto noiseUniform(std::ptrdiff_t width, std::ptrdiff_t height, typename
↪  Pixel::channel_type min, typename Pixel::channel_type max);
```

Preconditions: `width` ≥ 0, `height` ≥ 0, `max` $\geq$ `min`.
Returns: An image view of size `width` $*$ `height` with pixels generated using a pseudo-random number generator. The generator will generate every component of the pixels using uniform distribution.

8.1.10 `mgil::fromRange`

```
template<typename Range, typename Pixel =
↪  details::infer_std_range_over_pixel_t<std::remove_cvref_t<Range>>>
constexpr auto fromRange(Range &&pixels, std::ptrdiff_t width, std::ptrdiff_t
↪  height);
```

Preconditions: `width` ≥ 0, `height` ≥ 0, `std::ranges::size(pixels)` $==$ `width` $*$ `height`.
Returns: An image view of size `width` $*$ `height` generated by mapping the content of the *SizedRange* `pixels`.

```
template<typename Range, typename Pixel =
↪  details::infer_std_range_over_pixel_t<std::remove_cvref_t<Range>>>
constexpr auto fromRange(Range &&pixels);
```

Preconditions: All nested ranges of `pixels` have same size.
Returns: An image view of size
`std::ranges::size(pixels)` $*$ `std::ranges::size(*std::ranges::begin(pixels))` generated by mapping the content of the nested *ForwardRange* `pixels`.

8.2 Image View Adaptors

8.2.1 `mgil::crop`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto crop(View view, std::ptrdiff_t x, std::ptrdiff_t y, std::ptrdiff_t
↪  width, std::ptrdiff_t height);
```

Preconditions: $0 \leq$ `x` $\leq$ `view.width()`, $0 \leq$ `y` $\leq$ `view.height()`, $0 \leq$ `x`$+$`width` $\leq$ `view.width()`, $0 \leq$ `y` $+$ `height` $\leq$ `view.height()`.
Returns: An image view of size `width` $*$ `height` generated by mapping the content of the *SizedRange* `pixels`.

8.2.2 `mgil::flipHorizontal`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto flipHorizontal(View view);
```

Preconditions: None.
Returns: An image view generated by flipping `view` horizontally.

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto flipVertical(View view);
```

Preconditions: None.
Returns: An image view generated by flipping `view` vertically.

8.2.3 `mgil::rotate90`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto rotate90(View view);
```

Preconditions: None.
Returns: An image view generated by rotating `view` 90° clockwise.

8.2.4 `mgil::rotate180`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto rotate180(View view);
```

Preconditions: None.
Returns: An image view generated by rotating `view` 180° clockwise.

8.2.5 `mgil::rotate270`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto rotate270(View view);
```

Preconditions: None.
Returns: An image view generated by rotating `view` 270° clockwise.

8.2.6 `mgil::channelExtract`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel_> and IsImageView<View>
constexpr auto channelExtract(View view, std::size_t const index);
```

Preconditions: The pixel type of `view` has channel specified by `index`.
Returns: An grayscale image view generated by extracting the channel specified by `index`.

8.2.7 `mgil::transform`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto transform(View view, auto const &func);
```

Preconditions: The pixel type of `view` has channel specified by `index`.
Returns: An image view with every pixels transformed by the function `func(pixel)`.

8.2.8 `mgil::colorConvert`

```
template<typename DstType, typename DstLayout, typename DstPixel = Pixel<DstType,
↪  DstLayout>, typename View, typename SrcPixel = typename View::value_type>
    requires IsPixel<SrcPixel> and IsPixel<DstPixel> and IsImageView<View> and
    ↪  IsPixelsColorConvertible<SrcPixel, DstPixel>
constexpr auto colorConvert(View view, DstType const &type_tag, DstLayout const
↪  &layout_tag);
```

Preconditions: None.
Returns: An image view generated by converting the original image view to the specified color space layout.

8.2.9 `mgil::nearest`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto nearest(View view, std::ptrdiff_t result_width, std::ptrdiff_t
↪  result_height);
```

Preconditions: `result_width` ≥ 0, `result_height` ≥ 0.
Returns: An image view resized to `result_width` $*$ `result_height` by nearest linear interpolation.

8.2.10 `mgil::padConstant`

```
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto padConstant(View view, std::ptrdiff_t pad_x, std::ptrdiff_t pad_y,
↪  Pixel const &pad_pixel);
```

Preconditions: `pad_x` ≥ 0, `pad_y` ≥ 0.
Returns: An image view obtained by padding the original image view by `pad_pixel`.

8.3 Image Algorithms

8.3.1 `mgil::convolve`

```
template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageContainer<Image>
constexpr auto convolve(View1 src, View2 kernel) -> Image;
```

Preconditions: `kernel.width()` $==$ `kernel.height()`, `kernel.width()`$\%2 == 0$
Returns: Convolution result of `src`.
Time Complexity: Let W be `src.width()`, H be `src.height()`, K be `kernel.width()`. The time complexity is $\mathcal{O}(W \cdot H \cdot K^2)$.

8.3.2 `mgil::boxBlur`

```
template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto boxBlur(View src, std::size_t const radius);
```

Preconditions: `radius` ≥ 0.
Returns: Applies a uniform blur using a box filter of size $(2r+1) * (2r+1)$.
Time Complexity: Let W be `src.width()`, H be `src.height()`, r be `radius`. The time complexity is $\mathcal{O}(W \cdot H \cdot r^2)$.

8.3.3 `mgil::gaussianBlur`

```
template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto gaussianBlur(View src, float const sigma) -> Image;
```

Preconditions: `sigma` > 0
Returns: Applies a Gaussian blur.
Time Complexity: Let W be `src.width()`, H be `src.height()`. The time complexity is $\mathcal{O}(W \cdot H)$.

8.3.4 `mgil::erode`

```
template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and std::same_as<typename
    ↪  View2::value_type::channel_type, bool>
constexpr auto erode(View1 src, View2 se) -> Image;
```

Preconditions: `se` must define a valid non-empty structuring element.
Returns: Erosion of the input image.
Time Complexity: Let W be `src.width()`, H be `src.height()`, $|$`se`$|$ be the number of non-zero elements in `se`. The time complexity is $\mathcal{O}(W \cdot H \cdot |$`se`$|)$.

8.3.5 `mgil::dilate`

```
template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and std::same_as<typename
    ↪  View2::value_type::channel_type, bool>
constexpr auto dilate(View1 src, View2 se) -> Image;
```

Preconditions: `se` must define a valid non-empty structuring element.
Returns: Dilation of the input image.
Time Complexity: Let W be `src.width()`, H be `src.height()`, $|$`se`$|$ be the number of non-zero elements in `se`. The time complexity is $\mathcal{O}(W \cdot H \cdot |$`se`$|)$.

8.3.6 `mgil::open`

```
template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and std::same_as<typename
    ↪  View2::value_type::channel_type, bool>
constexpr auto open(View1 src, View2 se) -> Image;
```

Preconditions: `se` must define a valid non-empty structuring element.
Returns: Performs morphological opening (erode then dilate).
Time Complexity: Let W be `src.width()`, H be `src.height()`, $|\texttt{se}|$ be the number of non-zero elements in `se`. The time complexity is $\mathcal{O}(W \cdot H \cdot |\texttt{se}|)$.

8.3.7 `mgil::close`

```
template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and std::same_as<typename
    ↪  View2::value_type::channel_type, bool>
constexpr auto close(View1 src, View2 se) -> Image;
```

Preconditions: `se` must define a valid non-empty structuring element.
Returns: Performs morphological closing (dilate then erode).
Time Complexity: Let W be `src.width()`, H be `src.height()`, $|\texttt{se}|$ be the number of non-zero elements in `se`. The time complexity is $\mathcal{O}(W \cdot H \cdot |\texttt{se}|)$.

8.3.8 `mgil::sobel`

```
template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto sobel(View src) -> Image;
```

Preconditions: None.
Returns: Gradient magnitude using the Sobel operator.
Time Complexity: Let W be `src.width()`, H be `src.height()`. The time complexity is $\mathcal{O}(W \cdot H)$.

8.4 Synopsis `mgil.hpp`

8.4.1 Namespace `mgil::details`

```
namespace mgil::details {
template<std::size_t Index, typename... Types>
struct nth_type;
template<std::size_t Index, typename... Types>
using nth_type_t = typename nth_type<Index, Types...>::type;

template<std::size_t Index, auto... Values>
struct nth_nttp;
template<std::size_t Index, auto... Values>
constexpr auto nth_nttp_v = nth_nttp<Index, Values...>::value;

template<typename... Types>
struct type_list;

template<std::size_t... indexes>
struct index_list;

template<typename T, template<auto...> class Templ>
struct is_specialization_of_nttp;
```

```cpp
template<typename T, template<auto...> class Templ>
constexpr bool is_specialization_of_nttp_v = is_specialization_of_nttp<T,
↪  Templ>::value;

template<typename T, template<typename...> class Templ>
struct is_specialization_of;

template<typename T, template<typename...> class Templ>
constexpr bool is_specialization_of_v = is_specialization_of<T, Templ>::value;

template<typename... Types>
struct sort_types;

template<typename... Types>
struct deduplicate_types;

template<typename TypeToFind, typename... Types>
struct find_type;

template<typename TypeToFind, typename... Types>
struct find_type<TypeToFind, type_list<Types...>> : find_type<TypeToFind,
↪  Types...> {};

template<typename TypeToFind, typename... Types>
constexpr std::size_t find_type_v = find_type<TypeToFind, Types...>::value;

template<typename TypeToFind, typename... Types>
struct contain_type;

template<typename TypeToFind, typename... Types>
constexpr bool contain_type_v = contain_type<TypeToFind, Types...>::value;

template<typename TypeList, typename IndexList>
struct rearrange_type_list;

template<typename TypeList, typename IndexList>
using rearrange_type_list_t = typename rearrange_type_list<TypeList,
↪  IndexList>::type;

struct image_adaptor_closure_tag {};

inline namespace concepts {
    template<typename Lhs, typename Rhs, typename Image>
    concept PipeInvocable;

    template<typename Type>
    concept IsImageAdaptorClosure;

    template<typename Adaptor, typename... Args>
    concept AdaptorInvocable;
} // namespace concepts

template<typename T, typename U>
using like_t = decltype(std::forward_like<T>(std::declval<U>()));

template<typename Lhs, typename Rhs>
struct Pipe : image_adaptor_closure_tag {
    [[no_unique_address]] Lhs _lhs;
    [[no_unique_address]] Rhs _rhs;

    template<typename Tp, typename Up>
```

```
    constexpr Pipe(Tp &&lhs, Up &&rhs);

#if __cpp_explicit_this_parameter >= 202110L
    template<typename Self, typename Image>
        requires PipeInvocable<like_t<Self, Lhs>, like_t<Self, Rhs>, Image>
    constexpr auto operator()(this Self &&self, Image &&img);
#elif
    template<typename Image>
        requires PipeInvocable<Lhs const &, Rhs const &, Image>
    constexpr auto operator()(Image &&img) const &;

    template<typename Image>
        requires PipeInvocable<Lhs &&, Rhs &&, Image>
    constexpr auto operator()(Image &&img) &&;

    template<typename Image>
    constexpr auto operator()(Image &&img) const && =
            DELETE_DEF("Invoking a pipe object with const r-value references is
            ↪  prohibited");
#endif
};

template<typename Fn>
struct pipeable : Fn, image_adaptor_closure_tag;

template<typename Self, typename Image>
    requires IsImageAdaptorClosure<Self> and AdaptorInvocable<Self, Image>
constexpr auto operator|(Image &&img, Self &&self);

template<typename Lhs, typename Rhs>
    requires IsImageAdaptorClosure<Lhs> and IsImageAdaptorClosure<Rhs>
constexpr auto operator|(Lhs &&lhs, Rhs &&rhs);

template<typename T>
concept IsArithmetic;

template<typename OutPtr, typename In>
constexpr auto ptr_reinterpret_cast(In *ptr) -> OutPtr;

template<typename OutPtr, typename In>
constexpr auto ptr_reinterpret_cast_const(In *ptr) -> OutPtr;

template<class Reference>
struct arrow_proxy;

template<typename T>
concept ImplsDistanceTo;

template<typename>
struct infer_difference_type;
template<typename T>
using infer_difference_type_t = typename infer_difference_type<T>::type;

template<typename T>
struct infer_value_type;
template<typename T>
using infer_value_type_t = typename infer_value_type<T>::type;

template<typename T>
concept ImplsIncrement;
template<typename T>
```

```
concept ImplsDecrement;
template<typename T>
concept ImplsAdvance;
template<typename T>
concept ImplsEqualTo;
template<typename T>
concept IsRandomAccessIterator;
template<typename T>
concept IsBidirectionalIterator;
template<typename T>
concept IsSinglePassIterator;
template<typename Arg, typename It>
concept DifferenceTypeArg;

template<typename Derived>
class iterator_facade {
public:
    using self_type = Derived;

private:
    auto self() -> self_type &;
    auto self() const -> self_type const &;

public:
    decltype(auto) operator*() const;
    auto operator->() const;

    friend auto operator==(self_type const &lhs, self_type const &rhs) -> bool;

    auto operator++() -> self_type &;
    auto operator++(int) -> self_type;
    auto operator--() -> self_type &;
    auto operator--(int) -> self_type;

    friend auto operator+=(self_type &self, DifferenceTypeArg<self_type> auto
    ↪  offset) -> self_type &
        requires ImplsAdvance<self_type>;
    friend auto operator+(self_type self, DifferenceTypeArg<self_type> auto
    ↪  offset) -> self_type
        requires ImplsAdvance<self_type>;
    friend auto operator+(DifferenceTypeArg<self_type> auto offset, self_type
    ↪  self) -> self_type
        requires ImplsAdvance<self_type>;
    friend auto operator-(self_type self, DifferenceTypeArg<self_type> auto
    ↪  offset) -> self_type
        requires ImplsAdvance<self_type>;
    friend auto operator-=(self_type &self, DifferenceTypeArg<self_type> auto
    ↪  offset) -> self_type &
        requires ImplsAdvance<self_type>;
    decltype(auto) operator[](DifferenceTypeArg<self_type> auto offset)
        requires ImplsAdvance<self_type>;
    decltype(auto) operator[](DifferenceTypeArg<self_type> auto offset) const
        requires ImplsAdvance<self_type>;
    friend auto operator-(self_type const &lhs, self_type const &rhs)
        requires ImplsDistanceTo<self_type>;
    friend auto operator<=>(self_type const &lhs, self_type const &rhs)
        requires ImplsDistanceTo<self_type>;
};

template<typename Loc, typename XIt, typename YIt, typename PointType>
```

```
    requires std::same_as<typename PointType::value_type, typename
    ↪  std::iterator_traits<XIt>::difference_type>
class position_locator_base {
public:
    using x_iterator = XIt;
    using y_iterator = YIt;

    using value_type = typename std::iterator_traits<x_iterator>::value_type;
    using reference = typename std::iterator_traits<x_iterator>::reference;
    using coordinate_type = typename
    ↪  std::iterator_traits<x_iterator>::difference_type;
    using difference_type = PointType;
    using point_type = PointType;
    using x_coordinate_type = coordinate_type;
    using y_coordinate_type = coordinate_type;
    using cached_location_type = difference_type;

private:
    constexpr auto self() -> Loc &;
    constexpr auto self() const -> Loc const &;

public:
    constexpr auto operator==(Loc const &that) const -> bool;
    constexpr auto operator!=(Loc const &that) const -> bool;

    constexpr auto xAt(x_coordinate_type dx, y_coordinate_type dy) const ->
    ↪  x_iterator;
    constexpr auto xAt(difference_type d) const -> x_iterator;
    constexpr auto yAt(x_coordinate_type dx, y_coordinate_type dy) const ->
    ↪  x_iterator;
    constexpr auto yAt(difference_type d) const -> x_iterator;
    constexpr auto xyAt(x_coordinate_type dx, y_coordinate_type dy) const -> Loc;
    constexpr auto xyAt(difference_type d) const -> Loc;

    constexpr auto operator()(x_coordinate_type dx, y_coordinate_type dy) const
    ↪  -> reference;
    constexpr auto operator[](difference_type const &d) const -> reference;

    constexpr auto operator*() const -> reference;

    constexpr auto operator+=(difference_type const &d) -> Loc &;
    constexpr auto operator-=(difference_type const &d) const -> Loc &;

    constexpr auto operator+(difference_type const &d) const -> Loc;
    constexpr auto operator-(difference_type const &d) const -> Loc;

    constexpr auto cacheLocation(difference_type const &d) const ->
    ↪  cached_location_type;
    constexpr auto cacheLocation(x_coordinate_type x, y_coordinate_type y) const
    ↪  -> cached_location_type;

    template<typename Deref, bool IsTransposed, typename PointType_>
    friend class position_locator;
};

template<typename Range>
struct infer_std_range_over_pixel;

template<typename Range>
using infer_std_range_over_pixel_t = typename
↪  infer_std_range_over_pixel<Range>::type;
```

```
} // namespace mgil::details
```

8.4.2 Namespace `mgil::traits`

```
namespace mgil::inline traits {
template<typename T>
struct ChannelTraitsImpl;

struct channel_tag;

template<typename T>
struct ChannelTraits : ChannelTraitsImpl<T>;

struct pixel_iterator_tag;

template<typename T>
struct PixelIteratorTraits;

template<typename T>
struct PixelTraits;
} // namespace mgil::inline traits
```

8.4.3 Namespace `mgil::concepts`

```
namespace mgil::inline concepts {
template<typename T>
concept IsPoint;;

template<typename T>
concept IsChannel;

template<typename T, typename U>
concept IsChannelsCompatible;

template<typename Src, typename Dst>
concept IsChannelsConvertible;

template<typename T>
concept IsColorSpace;

struct color_space_component_tag;

template<typename T>
concept IsColorSpaceComponent;

template<typename ColorSpace_, typename ChannelMapping_>
    requires details::is_specialization_of_nttp_v<ChannelMapping_,
    ↪  details::index_list> and
    details::is_specialization_of_v<ColorSpace_, details::type_list> and
    (ColorSpace_::template allConformTo<[]<typename T>() { return
    ↪  IsColorSpaceComponent<T>; }>()) and
    (ColorSpace_::size == ChannelMapping_::size)
struct ColorSpaceLayout {
    using color_space = ColorSpace_;
    using channel_mapping = ChannelMapping_;
    using mapped_color_space = details::rearrange_type_list_t<ColorSpace_,
    ↪  ChannelMapping_>;
};

template<typename T>
concept IsLayout;
```

```
template<typename T, typename U>
concept IsLayoutCompatible;

template<typename T>
concept IsPixel;

template<typename T, typename U>
concept IsPixelsCompatible;

template<typename T, typename U>
concept IsPixelsConvertible;

template<typename T, typename U>
concept IsPixelsColorConvertible;

template<typename It>
concept IsPixelIterator;

template<typename Pixel>
    requires IsPixel<Pixel>
struct identity_deref_adaptor {
    using value_type = Pixel;
    using const_adaptor = identity_deref_adaptor;
};

template<typename T>
concept IsPixelDereferenceAdaptor;

template<typename ConstT, typename Value, typename Reference, typename
↪  ConstReference, typename ResultType,
           bool IsMutable>
struct deref_base {
    using result_type = ResultType;
    using const_adaptor = ConstT;
    using value_type = Value;
    using reference = Reference;
    using const_reference = ConstReference;
    static constexpr bool is_mutable = IsMutable;
};
template<typename D1, typename D2>
    requires IsPixelDereferenceAdaptor<D1> and IsPixelDereferenceAdaptor<D2>
class deref_compose
    : public
        deref_base<deref_compose<typename D1::const_adaptor, typename
        ↪  D2::const_adaptor>,
        typename D1::value_type, typename D1::reference, typename
        ↪  D1::const_reference,
        typename D1::result_type, D1::is_mutable and D2::is_mutable> {
public:
    D1 fn1_;
    D2 fn2_;

    using result_type = typename D1::result_type;

    deref_compose();
    deref_compose(D1 const &d1, D2 const &d2);
    deref_compose(deref_compose const &that);

    template<typename D1_, typename D2_>
    explicit deref_compose(deref_compose<D1_, D2_> const &that);
```

```
    auto operator()(auto x) const;
    auto operator()(auto x);
};

template<typename It>
concept IsStepIterator;
template<typename It>
concept IsMutableStepIterator;

template<typename It>
concept IsMemoryBasedIterator;

template<typename It>
concept IsPixelIteratorHasTransposedType;

template<typename Loc>
concept Is2DLocator;

template<typename Loc>
concept IsPixelLocator;

template<typename Loc>
concept IsPixelLocatorHasTransposedType;

template<typename View>
concept IsImageView;

template<typename V1, typename V2>
concept IsImageViewsCompatible;

template<typename T>
concept IsImageContainer;

template<typename T>
concept IsImageFileIOClass;
} // namespace mgil::inline concepts
```

8.4.4 Namespace `mgil`

```
namespace mgil {
struct red_color_t;
struct green_color_t;
struct blue_color_t;
struct alpha_color_t;
struct gray_color_t;
struct cyan_color_t;
struct magenta_color_t;
struct yellow_color_t;
struct black_color_t;

using gray_t = details::type_list<gray_color_t>;
using rgb_t = details::type_list<red_color_t, green_color_t, blue_color_t>;
using rgba_t = details::type_list<red_color_t, green_color_t, blue_color_t,
↪  alpha_color_t>;
using cmyk_t = details::type_list<cyan_color_t, magenta_color_t, yellow_color_t,
↪  black_color_t>;

using gray_layout_t = ColorSpaceLayout<gray_t, details::index_list<0>>;
using rgb_layout_t = ColorSpaceLayout<rgb_t, details::index_list<0, 1, 2>>;
using bgr_layout_t = ColorSpaceLayout<rgb_t, details::index_list<2, 1, 0>>;
```

```
using rgba_layout_t = ColorSpaceLayout<rgba_t, details::index_list<0, 1, 2, 3>>;
using bgra_layout_t = ColorSpaceLayout<rgba_t, details::index_list<2, 1, 0, 3>>;
using abgr_layout_t = ColorSpaceLayout<rgba_t, details::index_list<3, 2, 1, 0>>;
using argb_layout_t = ColorSpaceLayout<rgba_t, details::index_list<3, 0, 1, 2>>;
using cmyk_layout_t = ColorSpaceLayout<cmyk_t, details::index_list<0, 1, 2, 3>>;

template<IsChannel T, typename ChannelTraits<T>::value_type Min, typename
↪  ChannelTraits<T>::value_type Max>
struct RescopedChannel;

template<IsChannel T, typename ChannelTraits<T>::value_type Min, typename
↪  ChannelTraits<T>::value_type Max>
    requires std::integral<T> or std::floating_point<T>
struct RescopedChannel<T, Min, Max> : channel_tag {
    using value_type = T;
    using reference = T &;
    using pointer = T *;
    using const_reference = T const &;
    using const_pointer = T const *;
    static constexpr bool is_mutable = true;
    constexpr RescopedChannel();
    constexpr RescopedChannel(RescopedChannel const &that);
    constexpr RescopedChannel(RescopedChannel &&that) noexcept;
    constexpr auto operator=(RescopedChannel const &that) -> RescopedChannel &;
    constexpr auto operator=(RescopedChannel &&that) noexcept -> RescopedChannel
    ↪  &;
    explicit constexpr RescopedChannel(T value_);
    template<typename U>
        requires std::convertible_to<U, T>
    explicit constexpr RescopedChannel(U value_);
    static constexpr auto minValue() -> value_type;
    static constexpr auto maxValue() -> value_type;
    [[nodiscard]] constexpr auto getValue() const -> value_type;
    constexpr auto setValue(value_type v) -> void;

    template<typename U>
        requires std::convertible_to<U, value_type>
    constexpr auto operator=(U const &that) -> RescopedChannel &;

    constexpr operator value_type() const;

    friend constexpr auto operator+(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs);
    friend constexpr auto operator-(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs);
    friend constexpr auto operator*(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs);
    friend constexpr auto operator/(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs);
    friend constexpr auto operator+(RescopedChannel const &lhs, value_type const
    ↪  &rhs);
    friend constexpr auto operator-(RescopedChannel const &lhs, value_type const
    ↪  &rhs);
    friend constexpr auto operator*(RescopedChannel const &lhs, value_type const
    ↪  &rhs);
    friend constexpr auto operator/(RescopedChannel const &lhs, value_type const
    ↪  &rhs);
    friend constexpr auto operator+(value_type const &lhs, RescopedChannel const
    ↪  &rhs);
    friend constexpr auto operator-(value_type const &lhs, RescopedChannel const
    ↪  &rhs);
```

```
    friend constexpr auto operator*(value_type const &lhs, RescopedChannel const
    ↪  &rhs);
    friend constexpr auto operator/(value_type const &lhs, RescopedChannel const
    ↪  &rhs);

    constexpr auto operator+=(auto const &rhs) -> RescopedChannel &;
    constexpr auto operator-=(auto const &rhs) -> RescopedChannel &;
    constexpr auto operator*=(auto const &rhs) -> RescopedChannel &;
    constexpr auto operator/=(auto const &rhs) -> RescopedChannel &;

    friend constexpr auto operator<=>(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs)
            -> std::strong_ordering;
    friend constexpr auto operator==(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs) -> bool;
    friend constexpr auto operator!=(RescopedChannel const &lhs, RescopedChannel
    ↪  const &rhs) -> bool;

    friend constexpr auto operator<<(std::ostream &os, RescopedChannel const
    ↪  &rhs) -> std::ostream &;
};

template<IsChannel T, typename ChannelTraits<T>::value_type Min, typename
↪  ChannelTraits<T>::value_type Max>
    requires std::is_class_v<T>
struct RescopedChannel<T, Min, Max> : T {
    static constexpr auto minValue() -> typename ChannelTraits<T>::value_type;
    static constexpr auto maxValue() -> typename ChannelTraits<T>::value_type;
    [[nodiscard]] constexpr auto getValue() const -> typename
    ↪  ChannelTraits<T>::value_type;
    constexpr auto setValue(typename ChannelTraits<T>::value_type v) -> void;
};

using Float_01 = RescopedChannel<float, 0.0f, 1.0f>;
using Double_01 = RescopedChannel<double, 0.0, 1.0>;
using UInt8_0255 = RescopedChannel<std::uint8_t, 0, 255>;
using UInt16_0255 = RescopedChannel<std::uint16_t, 0, 255>;
using UInt32_0255 = RescopedChannel<std::uint32_t, 0, 255>;
using UInt64_0255 = RescopedChannel<std::uint64_t, 0, 255>;

template<typename ChannelType, typename Layout>
    requires IsChannel<ChannelType> and IsLayout<Layout>
class Pixel {
public:
    using layout_type = Layout;
    using channel_type = ChannelType;
    using value_type = Pixel;
    using reference = value_type &;
    using const_reference = value_type const &;

private:
    static constexpr std::size_t size = Layout::color_space::size;

public:
    Pixel();
    Pixel(Pixel const &);
    Pixel(Pixel &&) noexcept(std::is_nothrow_move_constructible_v<ChannelType>);
    auto operator=(Pixel const &) -> Pixel &;
    auto operator=(Pixel &&)
    ↪  noexcept(std::is_nothrow_move_assignable_v<ChannelType>) -> Pixel &;
```

```
    explicit constexpr Pixel(ChannelType v);

    template<typename... Ts>
        requires(std::convertible_to<Ts, ChannelType> && ...) and (sizeof...(Ts)
        ↪  == size)
    explicit constexpr Pixel(Ts... components);
    explicit constexpr Pixel(std::array<ChannelType, size> const &arr);

    template<typename Range>
        requires std::ranges::range<Range> and
        ↪  std::convertible_to<std::ranges::range_value_t<Range>, ChannelType>
    explicit constexpr Pixel(Range &&range);

    template<typename SrcPixel>
        requires IsPixelsConvertible<SrcPixel, Pixel>
    explicit constexpr Pixel(SrcPixel const &src);

    template<typename SrcPixel, typename DstPixel>
        requires IsPixelsConvertible<SrcPixel, DstPixel>
    static constexpr auto convertPixelTo(SrcPixel const &src) -> DstPixel;

    template<typename Channel>
        requires IsChannel<Channel>
    constexpr auto castTo() const -> Pixel<Channel, layout_type>;

    friend constexpr auto operator<=>(Pixel const &lhs, Pixel const &rhs) ->
    ↪  std::strong_ordering;
    friend constexpr auto operator==(Pixel const &lhs, Pixel const &rhs) -> bool;
    friend constexpr auto operator!=(Pixel const &lhs, Pixel const &rhs) -> bool;

    friend constexpr auto operator+(Pixel const &lhs, Pixel const &rhs) -> Pixel;
    friend constexpr auto operator-(Pixel const &lhs, Pixel const &rhs) -> Pixel;
    friend constexpr auto operator*(Pixel const &lhs, Pixel const &rhs) -> Pixel;
    friend constexpr auto operator/(Pixel const &lhs, Pixel const &rhs) -> Pixel;
    friend constexpr auto operator+(Pixel const &lhs,
    ↪  std::convertible_to<channel_type> auto rhs) -> Pixel;
    friend constexpr auto operator-(Pixel const &lhs,
    ↪  std::convertible_to<channel_type> auto rhs) -> Pixel;
    friend constexpr auto operator*(Pixel const &lhs,
    ↪  std::convertible_to<channel_type> auto rhs) -> Pixel;
    friend constexpr auto operator/(Pixel const &lhs,
    ↪  std::convertible_to<channel_type> auto rhs) -> Pixel;
    friend constexpr auto operator+(std::convertible_to<channel_type> auto lhs,
    ↪  Pixel const &rhs) -> Pixel;
    friend constexpr auto operator-(std::convertible_to<channel_type> auto rhs,
    ↪  Pixel const &lhs) -> Pixel;
    friend constexpr auto operator*(std::convertible_to<channel_type> auto lhs,
    ↪  Pixel const &rhs) -> Pixel;
    friend constexpr auto operator/(std::convertible_to<channel_type> auto lhs,
    ↪  Pixel const &rhs) -> Pixel;

    constexpr auto operator+=(auto rhs) -> Pixel &;
    constexpr auto operator-=(auto rhs) -> Pixel &;
    constexpr auto operator*=(auto rhs) -> Pixel &;
    constexpr auto operator/=(auto rhs) -> Pixel &;

    [[nodiscard]] constexpr auto get(std::size_t index) noexcept -> ChannelType
    ↪  &;
    [[nodiscard]] constexpr auto get(std::size_t index) const noexcept ->
    ↪  ChannelType const &;
    template<std::size_t Index>
```

```
        requires(Index < size)
    [[nodiscard]] constexpr auto get() noexcept -> ChannelType &;
    template<std::size_t Index>
        requires(Index < size)
    [[nodiscard]] constexpr auto get() const noexcept -> ChannelType const &;
    template<std::size_t Index>
    [[nodiscard]] constexpr auto getSemantic() noexcept -> ChannelType &;
    template<std::size_t Index>
    [[nodiscard]] constexpr auto getSemantic() const noexcept -> ChannelType
    ↪  const &;
    template<typename Component>
        requires IsColorSpaceComponent<Component> and
                    details::contain_type_v<Component, typename
                    ↪  layout_type::color_space>
    [[nodiscard]] constexpr auto get() noexcept -> ChannelType &;
    template<typename Component>
        requires IsColorSpaceComponent<Component> and
                    details::contain_type_v<Component, typename
                    ↪  layout_type::color_space>
    [[nodiscard]] constexpr auto get() const noexcept -> ChannelType const &;
    constexpr auto set(std::size_t index, ChannelType v) -> void;
    constexpr auto operator[](std::size_t const index) const noexcept ->
    ↪  ChannelType const &;
    constexpr auto operator[](std::size_t const index) -> ChannelType &;
    static constexpr auto getSize() noexcept -> std::size_t;

    friend auto operator<<(std::ostream &out, Pixel const &pixel) -> std::ostream
    ↪  &;
};

template<std::size_t Index, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> const &pixel) noexcept -> Channel const
↪  &;
template<std::size_t Index, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> &pixel) noexcept -> Channel &;
template<std::size_t Index, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> const &&pixel) noexcept -> Channel
↪  const &&;
template<std::size_t Index, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> &&pixel) noexcept -> Channel &&;
template<typename Component, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> const &pixel) noexcept -> Channel const
↪  &;
template<typename Component, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> &pixel) noexcept -> Channel &;
template<typename Component, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> const &&pixel) noexcept -> Channel
↪  const &&;
template<typename Component, typename Channel, typename Layout>
constexpr auto get(Pixel<Channel, Layout> &&pixel) noexcept -> Channel &&;

template<typename ValueType>
    requires std::is_arithmetic_v<ValueType>
class Point {
public:
    using value_type = ValueType;

    constexpr Point();
    constexpr Point(Point const &that);
    constexpr Point(Point &&that) noexcept;
    constexpr Point(value_type x, value_type y);
```

```
    constexpr Point(value_type v);
    constexpr auto operator=(Point const &that) -> Point &;
    constexpr auto operator=(Point &&that) noexcept -> Point &;

    constexpr auto x() noexcept -> value_type &;
    constexpr auto y() noexcept -> value_type &;
    [[nodiscard]] constexpr auto x() const noexcept -> value_type const &;
    [[nodiscard]] constexpr auto y() const noexcept -> value_type const &;

    friend constexpr auto operator+(Point const &lhs, Point const &rhs) -> Point;
    friend constexpr auto operator-(Point const &lhs, Point const &rhs) -> Point;
    friend constexpr auto operator*(Point const &lhs, Point const &rhs) -> Point;
    friend constexpr auto operator/(Point const &lhs, Point const &rhs) -> Point;
    friend constexpr auto operator+(Point const &lhs, details::IsArithmetic auto
    ↪  rhs) -> Point;
    friend constexpr auto operator-(Point const &lhs, details::IsArithmetic auto
    ↪  rhs) -> Point;
    friend constexpr auto operator*(Point const &lhs, details::IsArithmetic auto
    ↪  rhs) -> Point;
    friend constexpr auto operator/(Point const &lhs, details::IsArithmetic auto
    ↪  rhs) -> Point;
    friend constexpr auto operator+(details::IsArithmetic auto rhs, Point const
    ↪  &lhs) -> Point;
    friend constexpr auto operator-(details::IsArithmetic auto rhs, Point const
    ↪  &lhs) -> Point;
    friend constexpr auto operator*(details::IsArithmetic auto rhs, Point const
    ↪  &lhs) -> Point;
    friend constexpr auto operator/(details::IsArithmetic auto rhs, Point const
    ↪  &lhs) -> Point;

    friend constexpr auto operator+=(Point &lhs, Point const &rhs) -> Point &;
    friend constexpr auto operator-=(Point &lhs, Point const &rhs) -> Point &;
    friend constexpr auto operator*=(Point &lhs, Point const &rhs) -> Point &;
    friend constexpr auto operator/=(Point &lhs, Point const &rhs) -> Point &;
    friend constexpr auto operator+=(Point &lhs, details::IsArithmetic auto rhs)
    ↪  -> Point &;
    friend constexpr auto operator-=(Point &lhs, details::IsArithmetic auto rhs)
    ↪  -> Point &;
    friend constexpr auto operator*=(Point &lhs, details::IsArithmetic auto rhs)
    ↪  -> Point &;
    friend constexpr auto operator/=(Point &lhs, details::IsArithmetic auto rhs)
    ↪  -> Point &;

    friend constexpr auto operator==(Point const &lhs, Point const &rhs) -> bool;
    friend constexpr auto operator<=>(Point const &lhs, Point const &rhs) ->
    ↪  std::strong_ordering
        requires std::same_as<std::strong_ordering,
                              decltype(std::declval<value_type>() <=>
                              ↪  std::declval<value_type>())>;
    friend constexpr auto operator<=>(Point const &lhs, Point const &rhs) ->
    ↪  std::weak_ordering
        requires std::same_as<std::weak_ordering,
                              decltype(std::declval<value_type>() <=>
                              ↪  std::declval<value_type>())>;
    friend constexpr auto operator<=>(Point const &lhs, Point const &rhs) ->
    ↪  std::partial_ordering
        requires std::same_as<std::partial_ordering,
                              decltype(std::declval<value_type>() <=>
                              ↪  std::declval<value_type>())>;

    friend auto operator<<(std::ostream &os, Point const &rhs) -> std::ostream &;
```

```
};
template<std::size_t Index, typename ValueType>
    requires(Index == 0 or Index == 1)
constexpr auto get(Point<ValueType> const &point) -> ValueType const &;
template<std::size_t Index, typename ValueType>
    requires(Index == 0 or Index == 1)
constexpr auto get(Point<ValueType> &point) -> ValueType &;
template<std::size_t Index, typename ValueType>
    requires(Index == 0 or Index == 1)
constexpr auto get(Point<ValueType> const &&point) -> ValueType const &&;
template<std::size_t Index, typename ValueType>
    requires(Index == 0 or Index == 1)
constexpr auto get(Point<ValueType> &&point) -> ValueType &&;

template<typename Deref = identity_deref_adaptor<Pixel<int, gray_layout_t>>, bool
↪  IsTransposed = false,
           bool IsVirtual = true, typename PointType = Point<std::ptrdiff_t>>
    requires IsPoint<PointType> and IsPixelDereferenceAdaptor<Deref>
struct position_iterator : pixel_iterator_tag,
                           details::iterator_facade<position_iterator<Deref,
                           ↪  IsTransposed, IsVirtual, PointType>> {
public:
    using const_iterator = position_iterator const;
    using value_type = typename Deref::value_type;
    using difference_type = std::ptrdiff_t;
    using point_type = PointType;
    using deref_type = Deref;
    using transposed_type = position_iterator<Deref, not IsTransposed, IsVirtual,
    ↪  PointType>;
    using size_type = std::size_t;
    using x_coordinate_type = std::ptrdiff_t;
    using y_coordinate_type = std::ptrdiff_t;

    static constexpr bool is_mutable = true;
    static constexpr bool transposed = IsTransposed;

    template<typename NewDeref>
        requires IsPixelDereferenceAdaptor<NewDeref>
    struct add_deref {
        using type = position_iterator<deref_compose<NewDeref, Deref>,
        ↪  IsTransposed, IsVirtual, PointType>;
        static constexpr auto make(position_iterator const &it, NewDeref const
        ↪  &new_deref) -> type;
    };

    explicit constexpr position_iterator(PointType const &pos = {0, 0}, PointType
    ↪  const &step = {1, 1},
                                         Deref const &deref = {},
                                         ↪  x_coordinate_type const width =
                                         ↪  0,
                                         y_coordinate_type const height = 0,
                                         ↪  value_type *storage = nullptr);
    constexpr position_iterator(position_iterator const &that);

    constexpr auto operator=(position_iterator const &that) -> position_iterator
    ↪  &;

    constexpr auto pos() const -> PointType const &;
    constexpr auto pos() -> PointType &;
    constexpr auto step() const -> PointType const &;
    constexpr auto derefFn() const;
```

```
    constexpr auto setStep(difference_type step) -> void;

    constexpr auto dereference() const;
    constexpr auto increment() -> void;
    constexpr auto decrement() -> void;
    constexpr auto advance(difference_type d) -> void;
    constexpr auto distance_to(position_iterator const &that) const ->
    ↪  difference_type;
    constexpr auto equal_to(position_iterator const &that) const -> bool;

    constexpr auto width() -> x_coordinate_type &;
    constexpr auto height() -> y_coordinate_type &;
    [[nodiscard]] constexpr auto width() const -> x_coordinate_type const &;
    [[nodiscard]] constexpr auto height() const -> y_coordinate_type const &;
};

template<typename Loc>
    requires IsPixelLocator<Loc>
struct two_dimensional_iterator : pixel_iterator_tag,
↪  details::iterator_facade<two_dimensional_iterator<Loc>> {
public:
    using parent_type = details::iterator_facade<two_dimensional_iterator<Loc>>;
    using value_type = typename Loc::value_type;
    using difference_type = std::ptrdiff_t;
    using point_type = typename Loc::point_type;
    using size_type = std::size_t;
    using x_coordinate_type = std::ptrdiff_t;
    using y_coordinate_type = std::ptrdiff_t;
    using const_iterator = two_dimensional_iterator<typename Loc::const_locator>;

    static constexpr bool is_mutable = true;

    constexpr two_dimensional_iterator();
    explicit constexpr two_dimensional_iterator(Loc const &loc, x_coordinate_type
    ↪  const width,
                                                y_coordinate_type const height,
                                                ↪  x_coordinate_type const x,
                                                y_coordinate_type const y);
    constexpr two_dimensional_iterator(two_dimensional_iterator const &that);

    constexpr auto operator=(two_dimensional_iterator const &that) ->
    ↪  two_dimensional_iterator &;

    constexpr auto pos() const -> point_type const &;

    constexpr auto dereference() const;
    constexpr auto increment() -> void;
    constexpr auto decrement() -> void;
    constexpr auto advance(difference_type d) -> void;
    constexpr auto distance_to(two_dimensional_iterator const &that) const ->
    ↪  difference_type;
    constexpr auto equal_to(two_dimensional_iterator const &that) const -> bool;

    constexpr auto width() -> x_coordinate_type &;
    constexpr auto height() -> y_coordinate_type &;
    [[nodiscard]] constexpr auto width() const -> x_coordinate_type const &;
    [[nodiscard]] constexpr auto height() const -> y_coordinate_type const &;
};

template<typename Deref, bool IsTransposed = false, bool IsVirtual = true,
```

```
            typename PointType = Point<std::ptrdiff_t>>
    requires IsPixelDereferenceAdaptor<Deref> and IsPoint<PointType>
class position_locator
    : public details::position_locator_base
        <position_locator<Deref, IsTransposed, IsVirtual, PointType>,
        position_iterator<Deref, IsTransposed, IsVirtual, PointType>,
        position_iterator<Deref, IsTransposed, IsVirtual, PointType>,
        PointType> {
public:
    using parent_type =
            details::position_locator_base
            <position_locator<Deref, IsTransposed, IsVirtual, PointType>,
            position_iterator<Deref, IsTransposed, IsVirtual, PointType>,
            position_iterator<Deref, IsTransposed, IsVirtual, PointType>,
            ↪  PointType>;
    using deref_type = Deref;
    using const_locator = position_locator<typename Deref::const_adaptor,
    ↪  IsTransposed, IsVirtual, PointType>;
    using point_type = typename parent_type::point_type;
    using coordinate_type = typename parent_type::coordinate_type;
    using x_coordinate_type = typename parent_type::x_coordinate_type;
    using y_coordinate_type = typename parent_type::y_coordinate_type;
    using x_iterator = typename parent_type::x_iterator;
    using y_iterator = typename parent_type::y_iterator;
    using transposed_type = position_locator<Deref, not IsTransposed, IsVirtual,
    ↪  PointType>;
    using size_type = std::size_t;

    static constexpr bool is_transposed = IsTransposed;
    static constexpr bool is_mutable = true;

    template<typename NewDeref>
        requires IsPixelDereferenceAdaptor<NewDeref>
    struct add_deref {
        using type = position_locator<deref_compose<NewDeref, Deref>,
        ↪  IsTransposed, IsVirtual, PointType>;
        static constexpr auto make(position_locator const &it, NewDeref const
        ↪  &new_deref) -> type;
    };

    explicit constexpr position_locator(PointType const &pos = {0, 0}, PointType
    ↪  const &step = {1, 1},
                                        Deref const &deref = {},
                                        ↪  x_coordinate_type const width = 0,
                                        y_coordinate_type const height = 0,
                                        typename parent_type::value_type *storage
                                        ↪  = nullptr);
    constexpr position_locator(position_locator const &that);
    }
    constexpr auto operator=(position_locator const &that) -> position_locator &;
    constexpr ~position_locator() noexcept;

    template<typename Deref_, bool Transposed_, bool IsVirtual_, typename
    ↪  PointType_>
    explicit constexpr position_locator(position_locator<Deref_, Transposed_,
    ↪  IsVirtual_, PointType_> const &that);
    template<typename Deref_, bool Transposed_, bool IsVirtual_, typename
    ↪  PointType_>
    explicit constexpr position_locator(position_locator<Deref_, Transposed_,
    ↪  IsVirtual_, PointType_> const &that,
                                        coordinate_type y_step);
```

```
    template<typename Deref_, bool Transposed_, bool IsVirtual_, typename
    ↪  PointType_>
    explicit constexpr position_locator(position_locator<Deref_, Transposed_,
    ↪  IsVirtual_, PointType_> const &that,
                                        coordinate_type x_step, coordinate_type
                                        ↪  y_step, bool transpose = false);

    constexpr auto operator==(position_locator const &that) const -> bool;
    constexpr auto operator!=(position_locator const &that) const -> bool;

    constexpr auto x() -> x_iterator &;
    constexpr auto x() const -> x_iterator const &;
    constexpr auto y() -> y_iterator &;
    constexpr auto y() const -> y_iterator const &;

    constexpr auto pos() const -> point_type const &;
    constexpr auto pos() -> point_type &;
    constexpr auto step() const -> point_type const &;
    constexpr auto deref() const -> deref_type const &;

    constexpr auto width() -> x_coordinate_type &;
    constexpr auto height() -> y_coordinate_type &;
    [[nodiscard]] constexpr auto width() const -> x_coordinate_type const &;
    [[nodiscard]] constexpr auto height() const -> y_coordinate_type const &;

    constexpr auto is1DTraversable(x_coordinate_type) const -> bool;

    constexpr auto yDistanceTo(position_locator const &that, x_coordinate_type)
    ↪  const -> y_coordinate_type;
};

template<typename Loc>
    requires IsPixelLocator<Loc>
class ImageView {
public:
    using value_type = typename Loc::value_type;
    using reference = typename Loc::reference;
    using coordinate_type = typename Loc::coordinate_type;
    using difference_type = coordinate_type;
    using const_view = ImageView<typename Loc::const_locator>;
    using point_type = typename Loc::point_type;
    using locator = Loc;
    using iterator = two_dimensional_iterator<Loc>;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using size_type = std::size_t;
    using xy_locator = locator;
    using x_iterator = typename xy_locator::x_iterator;
    using y_iterator = typename xy_locator::y_iterator;
    using x_coordinate_type = typename xy_locator::x_coordinate_type;
    using y_coordinate_type = typename xy_locator::y_coordinate_type;

    template<typename Deref>
        requires IsPixelDereferenceAdaptor<Deref>
    struct add_deref {
        using type = ImageView<typename Loc::template add_deref<Deref>::type>;
        static auto make(ImageView const &view, Deref const &deref) -> type;
    };

    constexpr ImageView();
    template<typename View>
    explicit constexpr ImageView(View const &that);
```

```
    template<typename Loc_>
        requires IsPixelLocator<Loc_>
    constexpr ImageView(point_type const &dims, Loc_ const &loc);
    template<typename Loc_>
        requires IsPixelLocator<Loc_>
    constexpr ImageView(x_coordinate_type width, x_coordinate_type height, Loc_
    ↪  const &loc);

    template<typename View>
    constexpr auto operator=(View const &that) -> ImageView &;
    constexpr auto operator=(ImageView const &that) -> ImageView &;

    template<typename View>
    constexpr auto operator==(View const &that) const -> bool;
    template<typename View>
    constexpr auto operator!=(View const &that) const -> bool;

    constexpr auto width() const -> x_coordinate_type const &;
    constexpr auto height() const -> y_coordinate_type const &;
    constexpr auto pixels() const -> xy_locator const &;
    constexpr auto width() -> x_coordinate_type &;
    constexpr auto height() -> y_coordinate_type &;
    constexpr auto pixels() -> xy_locator &;

    [[nodiscard]] constexpr auto is1DTraversable() const -> bool;

    [[nodiscard]] constexpr auto empty() const -> bool;
    [[nodiscard]] constexpr auto size() const -> size_type;
    [[nodiscard]] constexpr auto begin() const -> iterator;
    [[nodiscard]] constexpr auto end() const -> iterator;
    [[nodiscard]] constexpr auto rbegin() const -> reverse_iterator;
    [[nodiscard]] constexpr auto rend() const -> reverse_iterator;
    [[nodiscard]] constexpr auto front() const -> reference;
    [[nodiscard]] constexpr auto back() const -> reference;

    [[nodiscard]] constexpr auto operator[](difference_type i) const ->
    ↪  reference;
    [[nodiscard]] constexpr auto at(difference_type i) const -> iterator;
    [[nodiscard]] constexpr auto at(point_type const &i) const -> iterator;
    [[nodiscard]] constexpr auto at(x_coordinate_type x, y_coordinate_type y)
    ↪  const -> iterator;

    [[nodiscard]] constexpr auto operator()(point_type const &p) const ->
    ↪  reference;
    [[nodiscard]] constexpr auto operator()(x_coordinate_type x,
    ↪  y_coordinate_type y) const -> reference;

    [[nodiscard]] constexpr auto xyAt(x_coordinate_type x, y_coordinate_type y)
    ↪  const -> xy_locator;
    [[nodiscard]] constexpr auto xyAt(point_type const &p) const -> xy_locator;

    [[nodiscard]] constexpr auto xAt(x_coordinate_type x, y_coordinate_type y)
    ↪  const -> x_iterator;
    [[nodiscard]] constexpr auto xAt(point_type const &p) const -> x_iterator;
    [[nodiscard]] constexpr auto rowBegin(y_coordinate_type y) const ->
    ↪  x_iterator;
    [[nodiscard]] constexpr auto rowEnd(y_coordinate_type y) const -> x_iterator;

    [[nodiscard]] constexpr auto yAt(x_coordinate_type x, y_coordinate_type y)
    ↪  const -> y_iterator;
    [[nodiscard]] constexpr auto yAt(point_type const &p) const -> y_iterator;
```

```
    [[nodiscard]] constexpr auto colBegin(x_coordinate_type x) const ->
    ↪  y_iterator;
    [[nodiscard]] constexpr auto colEnd(x_coordinate_type x) const -> y_iterator;

    friend auto operator<<(std::ostream &os, ImageView view) -> std::ostream &;
};

template<typename Pixel>
    requires IsPixel<Pixel>
class EmptyView {
    struct EmptyDeref : deref_base<EmptyDeref, Pixel, Pixel &, Pixel const &,
    ↪  Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &) const -> Pixel;
    };
    using locator = position_locator<EmptyDeref>;
    using point_type = Point<std::int64_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height) const
    ↪  -> view_type;
};
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto empty(std::ptrdiff_t width, std::ptrdiff_t height);

template<typename Pixel>
    requires IsPixel<Pixel>
class IdenticalView {
    struct IdenticalDeref : deref_base<IdenticalDeref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &) const -> Pixel;
    };
    using locator = position_locator<IdenticalDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height, Pixel
    ↪  const &pixel) const -> view_type;
};
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto identical(std::ptrdiff_t width, std::ptrdiff_t height, Pixel const
↪  &pixel);

template<typename Pixel>
    requires IsPixel<Pixel>
class GradientView {
    struct GradientDeref : deref_base<GradientDeref, Pixel, Pixel &, Pixel const
    ↪  &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<GradientDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height, Pixel
    ↪  const &start, Pixel const &step_x,
```

```
                                Pixel const &step_y) const -> view_type;
};
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto gradient(std::ptrdiff_t width, std::ptrdiff_t height, Pixel const
↪  &start, Pixel const &step_x,
                        Pixel const &step_y);

template<typename Pixel>
    requires IsPixel<Pixel>
class GenerateView {
    struct GenerateDeref : deref_base<GenerateDeref, Pixel, Pixel &, Pixel const
    ↪  &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<GenerateDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height, auto
    ↪  const &generator) const
            -> view_type;
};
template<typename Gen, typename Pixel = std::invoke_result_t<Gen, std::ptrdiff_t,
↪  std::ptrdiff_t>>
    requires IsPixel<Pixel>
constexpr auto generate(std::ptrdiff_t width, std::ptrdiff_t height, Gen
↪  generator);

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class PatternView {
    struct PatternDeref : deref_base<PatternDeref, Pixel, Pixel &, Pixel const &,
    ↪  Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<PatternDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, std::ptrdiff_t width, std::ptrdiff_t
    ↪  height) const -> view_type;
};
template<typename View, typename Pixel = typename View::value_type>
    requires IsPixel<Pixel> and IsImageView<View>
constexpr auto pattern(View view, std::ptrdiff_t width, std::ptrdiff_t height);

template<typename Pixel, typename View1, typename View2>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
class ConcatHorizontalView {
    struct ConcatHorizontalDeref : deref_base<ConcatHorizontalDeref, Pixel, Pixel
    ↪  &, Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<ConcatHorizontalDeref>;
```

```
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View1 view1, View2 view2) const;
};
template<typename View1, typename View2, typename Pixel = typename
↪  View1::value_type>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
constexpr auto concatHorizontal(View1 view1, View2 view2);

template<typename Pixel, typename View1, typename View2>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
class ConcatVerticalView {
    struct ConcatVerticalDeref : deref_base<ConcatVerticalDeref, Pixel, Pixel &,
    ↪  Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<ConcatVerticalDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View1 view1, View2 view2) const;
};
template<typename View1, typename View2, typename Pixel = typename
↪  View1::value_type>
    requires IsPixel<Pixel> and IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageViewsCompatible<View1, View2>
constexpr auto concatVertical(View1 view1, View2 view2);

template<typename Pixel>
    requires IsPixel<Pixel>
class CheckerboardView {
    struct CheckerboardDeref : deref_base<CheckerboardDeref, Pixel, Pixel &,
    ↪  Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<CheckerboardDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height, Pixel
    ↪  const &pixel1, Pixel const &pixel2,
                              std::ptrdiff_t cell_width, std::ptrdiff_t
                              ↪  cell_height) const;
};
template<typename Pixel>
    requires IsPixel<Pixel>
constexpr auto checkerboard(std::ptrdiff_t width, std::ptrdiff_t height, Pixel
↪  const &pixel1, Pixel const &pixel2,
                            std::ptrdiff_t cell_width, std::ptrdiff_t
                            ↪  cell_height);

template<typename Pixel, typename Gen>
    requires IsPixel<Pixel>
```

```
class NoiseUniformView {
    struct NoiseUniformDeref : deref_base<NoiseUniformDeref, Pixel, Pixel &,
    ↪  Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<NoiseUniformDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;
    using channel_type = typename Pixel::channel_type;

public:
    constexpr auto operator()(std::ptrdiff_t width, std::ptrdiff_t height,
    ↪  channel_type min,
                              channel_type max) const;
};
template<typename Pixel, typename Gen>
    requires IsPixel<Pixel>
constexpr auto noiseUniform(std::ptrdiff_t width, std::ptrdiff_t height, typename
↪  Pixel::channel_type min,
                            typename Pixel::channel_type max);

template<typename Range, typename Pixel =
↪  details::infer_std_range_over_pixel_t<Range>>
    requires IsPixel<Pixel> and
                (std::ranges::sized_range<Range> or
                (std::ranges::forward_range<Range> and
                std::ranges::range<std::ranges::range_value_t<Range>>))
class FromRangeView {
    struct FromRangeDeref : deref_base<FromRangeDeref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel
            requires(not std::ranges::range<std::ranges::range_value_t<Range>>);
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<FromRangeDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(Range &&pixels, std::ptrdiff_t width,
    ↪  std::ptrdiff_t height)
        requires(not std::ranges::range<std::ranges::range_value_t<Range>>);
    constexpr auto operator()(Range &&pixels)
        requires(std::ranges::range<std::ranges::range_value_t<Range>>);
};
template<typename Range, typename Pixel =
↪  details::infer_std_range_over_pixel_t<std::remove_cvref_t<Range>>>
constexpr auto fromRange(Range &&pixels, std::ptrdiff_t width, std::ptrdiff_t
↪  height);
template<typename Range, typename Pixel =
↪  details::infer_std_range_over_pixel_t<std::remove_cvref_t<Range>>>
constexpr auto fromRange(Range &&pixels);

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class CropView {
    struct CropDeref : deref_base<CropDeref, Pixel, Pixel &, Pixel const &,
    ↪  Pixel, false> {
```

```
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪ Pixel;
    };
    using locator = position_locator<CropDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, std::ptrdiff_t x, std::ptrdiff_t y,
    ↪ std::ptrdiff_t width,
                              std::ptrdiff_t height);
};
struct CropFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view, std::ptrdiff_t x, std::ptrdiff_t y,
    ↪ std::ptrdiff_t width,
                              std::ptrdiff_t height) const;
    constexpr auto operator()(std::ptrdiff_t x, std::ptrdiff_t y, std::ptrdiff_t
    ↪ width,
                              std::ptrdiff_t height) const;
};
inline constexpr auto crop = CropFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class FlipHorizontalView {
    struct FlipHorizontalDeref : deref_base<FlipHorizontalDeref, Pixel, Pixel &,
    ↪ Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪ Pixel;
    };
    using locator = position_locator<FlipHorizontalDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view);
};
struct FlipHorizontalFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view) const;
    constexpr auto operator()() const;
};
inline constexpr auto flipHorizontal = FlipHorizontalFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class FlipVerticalView {
    struct FlipVerticalDeref : deref_base<FlipVerticalDeref, Pixel, Pixel &,
    ↪ Pixel const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪ Pixel;
    };
    using locator = position_locator<FlipVerticalDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view);
};
struct FlipVerticalFn : details::image_adaptor_closure_tag {
```

```
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view) const;
    constexpr auto operator()() const;
};
inline constexpr auto flipVertical = FlipVerticalFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class Rotate90View {
    struct Rotate90Deref : deref_base<Rotate90Deref, Pixel, Pixel &, Pixel const
    ↪  &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<Rotate90Deref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view);
};
struct Rotate90Fn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view) const;
    constexpr auto operator()() const;
};
inline constexpr auto rotate90 = Rotate90Fn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class Rotate180View {
    struct Rotate180Deref : deref_base<Rotate180Deref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<Rotate180Deref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view);
};
struct Rotate180Fn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view) const;
    constexpr auto operator()() const;
};
inline constexpr auto rotate180 = Rotate180Fn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class Rotate270View {
    struct Rotate270Deref : deref_base<Rotate270Deref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<Rotate270Deref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;
```

```
public:
    constexpr auto operator()(View view);
};
struct Rotate270Fn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view) const;
    constexpr auto operator()() const;
};
inline constexpr auto rotate270 = Rotate270Fn{};

template<typename Pixel_, typename View>
    requires IsPixel<Pixel_> and IsImageView<View>
class ChannelExtractByIndexView {
protected:
    using channel_type = typename Pixel_::channel_type;
    using result_pixel_type = Pixel<channel_type, gray_layout_t>;
    struct ChannelExtractDeref : deref_base<ChannelExtractDeref,
    ↪  result_pixel_type, result_pixel_type &,
                                            result_pixel_type const &,
                                            ↪  result_pixel_type, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  result_pixel_type;
    };
    using locator = position_locator<ChannelExtractDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, std::size_t const index);
};
struct ChannelExtractByIndexFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view, std::size_t const index) const;
    constexpr auto operator()(std::size_t const index) const;
};
inline constexpr auto channelExtractByIndex = ChannelExtractByIndexFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class TransformView {
    struct TransformDeref : deref_base<TransformDeref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<TransformDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, auto const &func) const -> view_type;
};
struct TransformFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view, auto const &func) const;
    constexpr auto operator()(auto const &func) const;
};
inline constexpr auto transform = TransformFn{};

template<typename SrcPixel, typename DstPixel, typename View>
```

```
    requires IsPixel<SrcPixel> and IsPixel<DstPixel> and IsImageView<View> and
                 IsPixelsColorConvertible<SrcPixel, DstPixel>
class ColorConvertView {
    struct ColorConvertDeref
        : deref_base<ColorConvertDeref, DstPixel, DstPixel &, DstPixel const &,
        ↪  DstPixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  DstPixel;
    };
    using locator = position_locator<ColorConvertDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view) const -> view_type;
};
struct ColorConvertFn : details::image_adaptor_closure_tag {
    template<typename DstType, typename DstLayout, typename DstPixel =
    ↪  Pixel<DstType, DstLayout>, typename View,
                typename SrcPixel = typename View::value_type>
    constexpr auto operator()(View view, DstType const &type_tag, DstLayout const
    ↪  &layout_tag) const;
    template<typename DstType, typename DstLayout>
    constexpr auto operator()(DstType const &type_tag, DstLayout const
    ↪  &layout_tag) const;
};
inline constexpr auto colorConvert = ColorConvertFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class NearestView {
    struct NearestDeref : deref_base<NearestDeref, Pixel, Pixel &, Pixel const &,
    ↪  Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
    using locator = position_locator<NearestDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, std::ptrdiff_t result_width,
    ↪  std::ptrdiff_t result_height) const;
};
struct NearestFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view, std::ptrdiff_t result_width,
    ↪  std::ptrdiff_t result_height) const;
    constexpr auto operator()(std::ptrdiff_t result_width, std::ptrdiff_t
    ↪  result_height) const;
};
inline constexpr auto nearest = NearestFn{};

template<typename Pixel, typename View>
    requires IsPixel<Pixel> and IsImageView<View>
class PadConstantView {
    struct PadConstantDeref : deref_base<PadConstantDeref, Pixel, Pixel &, Pixel
    ↪  const &, Pixel, false> {
        constexpr auto operator()(Point<std::ptrdiff_t> const &pos) const ->
        ↪  Pixel;
    };
```

```
    using locator = position_locator<PadConstantDeref>;
    using point_type = Point<std::ptrdiff_t>;
    using view_type = ImageView<locator>;

public:
    constexpr auto operator()(View view, std::ptrdiff_t pad_x, std::ptrdiff_t
    ↪  pad_y, Pixel const &pad_pixel) const;
};
struct PadConstantFn : details::image_adaptor_closure_tag {
    template<typename View, typename Pixel = typename View::value_type>
    constexpr auto operator()(View view, std::ptrdiff_t pad_x, std::ptrdiff_t
    ↪  pad_y, Pixel const &pad_pixel) const;
    template<typename Pixel>
    constexpr auto operator()(std::ptrdiff_t pad_x, std::ptrdiff_t pad_y, Pixel
    ↪  const &pad_pixel) const;
};
inline constexpr auto padConstant = PadConstantFn{};

template<typename Pixel, typename Alloc = std::allocator<Pixel>>
    requires IsPixel<Pixel>
class Image {
public:
    using allocator_type = typename std::allocator_traits<Alloc>::template
    ↪  rebind_alloc<Pixel>;
    using locator = position_locator<identity_deref_adaptor<Pixel>, false,
    ↪  false>;
    using view_type = ImageView<locator>;
    using const_view_type = ImageView<typename locator::const_locator>;
    using point_type = typename view_type::point_type;
    using value_type = typename view_type::value_type;
    using x_coordinate_type = typename view_type::x_coordinate_type;
    using y_coordinate_type = typename view_type::y_coordinate_type;

    constexpr Image();
    constexpr Image(Image const &that);
    constexpr Image(Image &&that) noexcept;
    constexpr auto operator=(Image const &that) -> Image &;
    constexpr auto operator=(Image &&that) noexcept -> Image &;

    constexpr Image(x_coordinate_type width, y_coordinate_type height, Pixel
    ↪  const &initial = {},
                    allocator_type const alloc = {});
    template<typename Pixel_, typename Alloc_>
        requires IsPixelsCompatible<Pixel_, Pixel>
    explicit constexpr Image(Image<Pixel_, Alloc_> const &that);
    template<typename Pixel_, typename Alloc_>
        requires IsPixelsCompatible<Pixel_, Pixel>
    constexpr auto operator=(Image<Pixel_, Alloc_> const &that) -> Image &;

    template<typename View>
        requires IsImageView<std::remove_cvref_t<View>>
    explicit constexpr Image(View &&view);
    template<typename View>
        requires IsImageView<std::remove_cvref_t<View>>
    constexpr auto operator=(View &&view) -> Image &;

    constexpr auto operator==(Image const &that) const noexcept -> bool;
    constexpr auto operator[](x_coordinate_type x, y_coordinate_type y) const ->
    ↪  Pixel const &;
    constexpr auto operator[](x_coordinate_type x, y_coordinate_type y) -> Pixel
    ↪  &;
```

```
    constexpr auto operator[](point_type const &pos) const -> Pixel const &;
    constexpr auto operator[](point_type const &pos) -> Pixel &;
    [[nodiscard]] constexpr auto toView() const -> view_type;
};

template<typename View1, typename View2, typename Image = Image<typename
↪  View1::value_type>>
    requires IsImageView<View1> and IsImageView<View2> and
    ↪  IsImageContainer<Image>
constexpr auto convolve(View1 src, View2 kernel) -> Image;

template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto boxBlur(View src, std::size_t const radius);

template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto gaussianBlur(View src, float const sigma) -> Image;

template<typename View1, typename View2, typename Comparator, typename Image =
↪  Image<typename View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and
                std::same_as<typename View2::value_type::channel_type, bool>
constexpr auto morphologicalOperation(View1 src, View2 se, Comparator cmp) ->
↪  Image;

inline constexpr auto erode;
inline constexpr auto dilate;

template<typename View1, typename View2, typename Comparator, typename Image =
↪  Image<typename View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and
                std::same_as<typename View2::value_type::channel_type, bool>
constexpr auto open(View1 src, View2 se);
template<typename View1, typename View2, typename Comparator, typename Image =
↪  Image<typename View1::value_type>>
    requires IsImageView<View1> and IsImageContainer<Image> and
    ↪  IsImageView<View2> and
                std::same_as<typename View2::value_type::channel_type, bool>
constexpr auto close(View1 src, View2 se);

template<typename View, typename Image = Image<typename View::value_type>>
    requires IsImageView<View> and IsImageContainer<Image>
constexpr auto sobel(View src) -> Image;

template<typename Pixel = Pixel<UInt8_0255, rgb_layout_t>, typename Image =
↪  Image<Pixel>>
struct BMPFileIO {
    using value_type = Pixel;
    using image_type = Image;
#pragma pack(push, 1)
    struct BMPFileHeader {
        std::uint16_t type;
        std::uint32_t size;
        std::uint16_t reserved1;
        std::uint16_t reserved2;
        std::uint32_t offset;
    };
    static_assert(sizeof(BMPFileHeader) == 14, "BMPFileHeader size mismatch");
```

```
    struct BMPInfoHeader {
        std::uint32_t size;
        std::int32_t width;
        std::int32_t height;
        std::uint16_t planes;
        std::uint16_t bits_count;
        std::uint32_t compression;
        std::uint32_t image_size;
        std::int32_t x_pixels_per_m;
        std::int32_t y_pixels_per_m;
        std::uint32_t colors_used;
        std::uint32_t colors_important;
    };
    static_assert(sizeof(BMPInfoHeader) == 40, "BMPInfoHeader size mismatch");
#pragma pack(pop)

    enum class BMPFileIOError {
        FILE_NOT_FOUND,
        NOT_A_FILE,
        COULD_NOT_OPEN_FILE,
        FILE_READ_FAILED,
        FILE_WRITE_FAILED,
        HEADER_FORMAT_ERROR,
        FILE_FORMAT_ERROR,
        INFO_HEADER_ERROR,
        UNSUPPORTED
    };

    static auto readFile(std::filesystem::path const &image_path) ->
    ↪  std::expected<Image, BMPFileIOError>;
    template<typename View>
        requires IsImageView<View>
    static auto writeFile(View view, std::filesystem::path const &image_path)
            -> std::expected<void, BMPFileIOError>;
};

template<typename FileIO>
    requires IsImageFileIOClass<FileIO>
constexpr auto readImage(std::filesystem::path const &path);
template<typename FileIO, typename View>
    requires IsImageView<View> and IsImageFileIOClass<FileIO>
constexpr auto writeImage(View view, std::filesystem::path const &path);
} // namespace mgil
```