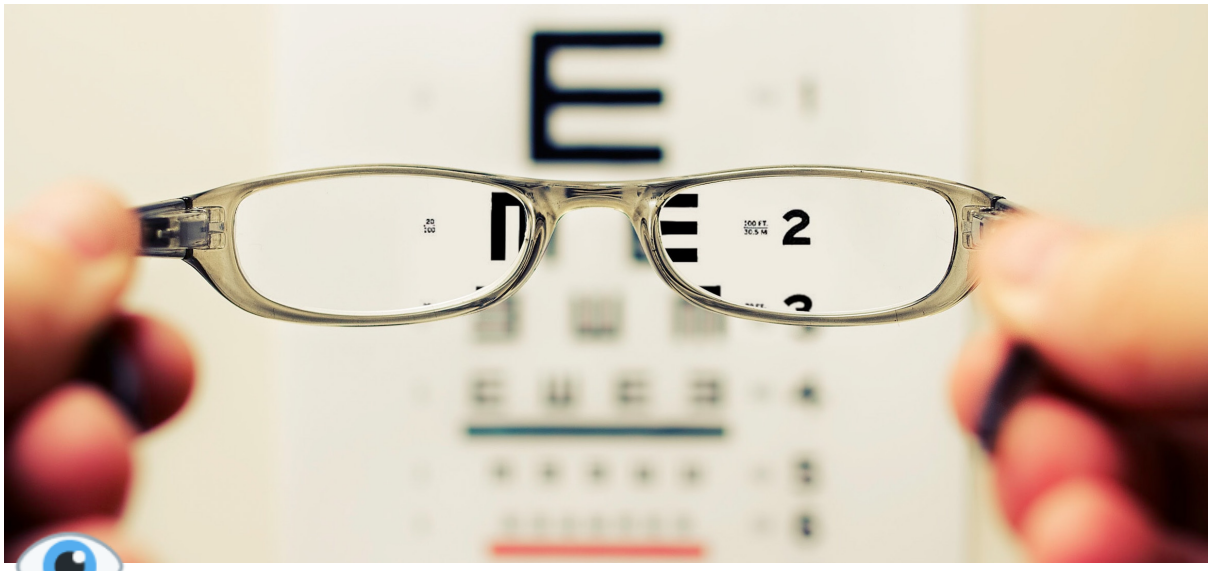

Implementation plan: 2d to 1d array

1.1 Name & Date

Dylan van Eck - 19/06/2020

1.2 Goal

The goal is to write an "Image Shell" for RGB and Intensity Images (greyscale). The "image shell" makes it possible to save the pixel values of RGB and Intensity images in a container. This container is useful for future edits.

1.3 Methods

A C++ container can be used to store arbitrary elements, such as integers or custom classes. These containers can be divided in four types of containers

Sequence containers

The common property of all sequential containers is that the elements can be accessed sequentially.

Associative containers

The elements are no longer ordered but instead have associations with each other used for determining uniqueness or mappings.

Unordered associative containers

The elements in the unordered associative containers are not ordered. This is due to the use of hashing to store objects. But the containers can still be iterated through like a regular associative container.

I will focus on the first one, because it's necessary to have the elements ordered for a picture.

Here's a quick summary of the sequence containers:

- `std::array` – static contiguous array, providing fast access but with a fixed number of elements
- `std::vector` – dynamic contiguous array, providing fast access but costly insertions/deletions
- `std::deque` – double-ended queue providing efficient insertion/deletion at the front and back of a sequence
- `std::list` and `std::forward_list` – linked list data structures, allowing for efficient insertion/deletion into the middle of a sequence

1.4. Choice

All kinds of associative containers are put together. I made checklists for each container and concluded from this what my best choice will be (see below). From these checklists, the std :: array works best and is also examined in my practicum.

`std::vector`

Your default sequential containers should be a `std::vector`.
Generally, `std::vector` will provide you with the right balance of performance and speed. The `std::vector` container is similar to a C-style array that can grow or shrink during runtime. The underlying buffer is stored contiguously and is guaranteed to be compatible with C-style arrays.

Consider using a std::vector if

Aa Name	☑ Check
You need your data to be stored contiguously in memory, Especially useful for C-style API compatibility	☑
You do not know the size at compile time	☐
You need efficient random access to your elements (O(1))	☑
You will be adding and removing elements from the end	☐
You want to iterate over the elements in any order	☑

Avoid using a std::vector if

Aa Name	☑ Check	☑ Maybe
You will frequently add or remove elements to the front or middle of the sequence	☐	☑
The size of your buffer is constant and known in advance (prefer std::array)	☑	☐

`std::array`

The `std::array` container is the most like a built-in array, but offering extra features such as bounds checking and automatic memory management. Unlike `std::vector`, the size of `std::array` is fixed and cannot change during runtime.

Consider using a std::array if

Aa Name	☑ Check
You need your data to be stored contiguously in memory, Especially useful for C-style API compatibility	☑
The size of your array is known in advance	☑
You need efficient random access to your elements (O(1))	☑
You want to iterate over the elements in any order	☑

Avoid using a std::array if

Aa Name	☑ Check	☑ Maybe

Aa Name	Check	Maybe
You need to insert or remove elements	☐	☑
You don't know the size of your array at compile time	☐	☐
You need to be able to resize your array dynamically	☐	☐

`std::deque`

The `std::deque` container gets its name from a shortening of "double ended queue". The `std::deque` container is most efficient when appending items to the front or back of a queue. Unlike `std::vector`, `std::deque` does not provide a mechanism to reserve a buffer. The underlying buffer is also not guaranteed to be compatible with C-style array APIs.

Consider using std::deque if

Aa Name	Check
You need to insert new elements at both the front and back of a sequence (e.g. in a scheduler)	☑
You need efficient random access to your elements (O(1))	☑
You want the internal buffer to automatically shrink when elements are removed	☐
You want to iterate over the elements in any order	☑

Avoid using std::deque if

Aa Name	Check	Maybe
You need to maintain compatibility with C-style APIs	☐	☐
You need to reserve memory ahead of time	☐	☐
You need to frequently insert or remove elements from the middle of the sequence. Calling insert in the middle of a std::deque invalidates all iterators and references to its elements.	☑	☐

`std::list`

The `std::list` and `std::forward_list` containers implement linked list data structures. Where `std::list` provides a doubly-linked list, the `std::forward_list` only contains a pointer to the next object. Unlike the

other sequential containers, the list types do not provide efficient random access to elements. Each element must be traversed in order.

Consider using std::list if

Aa Name	Check
You need to store many items but the number is unknown	☐
You need to insert or remove new elements from any position in the sequence	☑
You do not need efficient access to random elements	☐
You want the ability to move elements or sets of elements within the container or between different containers	☑
You want to implement a node-wise memory allocation scheme	☐

Avoid using std::list if

Aa Name	Check
You need to maintain compatibility with C-style APIs	☐
You need efficient access to random elements	☑
Your system utilizes a cache (prefer std::vector for reduced cache misses)	☐
The size of your data is known in advance and can be managed by a std::vector	☑

1.5 Implementation

The implementation of the RGB Image Shell is quiet simple:

1. An array is created in the RGBImageStudent.cpp with the size of the Image.
2. The array gets the pixel values assigned.

The implementation of the Intensity Image shell is done the same way as the IntensityImageStudent.cpp.

1.6 Evaluation

Because to default is using an 2d rgb pixelmap will I make this program to compare an 1-dimensional array with an 2-dimensional array based on the

amount of process time. To arrive at this comparison I am going to execute the formula (RGB → Intensity) 4000 times.

- 2000 times with a 1-dimensional array
- 2000 times with a 2-dimensional array

The process time will be calculated with the std::chrono::high_resolution_clock, and then divided by 2000 for each sort of array.

Usage of image

I will use all the three images of each type (so man-1, female-1, child-1). This will make a more accurate conclusion as opposed to using one image.

The implementation will be done round about 24 hours.