Augmentor Design Document

1. Introduction

`Augmentor` is a C++ library focused on image batch manipulation and processing. We would like to provide simple interface and excellent performance for users, so we have leveraged the following design ideas.

2. Declarative APIs

Similar as other data processing tools (e.g. Spark, Flink), our library also provides declarative APIs for users to build up their image manipulation pipelines. The advantage of declarative APIs is that the pipelines can be checked before the actual processing. The library evaluates the inputs of each operation, and make sure all of them are legal operations. For example, the input of `resize` operation cannot be negative. If invalid parameters are found, the program will be terminated in the building stage.

2.1. Build with chain operations

When users setup their pipelines, it is common to concatenate a set of operations to manipulate images. Thus, the pipeline builder of our library is designed to be chain-able. For example,

```
augmentor
.rotate(45,90,1)
.flip(HORIZONTAL, 1)
.crop(300, 300, true)
.resize(120,120,1)
.rapid_blur(5)
.invert(1);
```

In order to implement this style, we design the methods like

```
Augmentor& Augmentor::some_operation(parameter param...);
```

The `augmentor` has a vector to store the definitions of every operations, which looks like

```
std::vector<std::unique_ptr< Operation<Image> >> operations;
```

The reason to use `unique_ptr` will be discussed more in the Section 4.

2.2. Run the program

After building up a pipeline, users can run it simply by calling `sample` method. The current implementation is straight forward: for each image, loop through every operation and perform the transform on the image. It looks something like

```
for (auto &operation : operations) {
    image = operation->perform(image);
}
```

Since the processing of images are independent of one another, parallel programming will be used in future to speed up the processing.

3. Template

There are many formats of images. They share the same APIs (e.g. `getPixel`, `setPixel`), but have different implementations. It is preferred to generalize the library to manipulate images in different formats, so Template is used in this library. Here is an example:

```
template<typename Image>
class Operation {
//
// private content...
//
public:
    virtual Image* perform(Image* image) = 0;
};
```

Currently, this library only supports JPEG images.

```
// Image actually means JpegImage
Operation<Image> operations;
```

In future, we will introduce more image formats, like png, bmp.

```
Operation<PngImage> PngOperations;
Operation<BmpImage> BmpOperations;
```

3.1. Concept (in future)

This library is built under the C++17 standard. In future, we would like to upgrade to C++20, where Concept is introduced. Since images, despite their formats, share the same interface (e.g. `getHeight`, `getWidth`, `getPixel`, `setPixel`), it is recommended to constrain an image class using Concept. A concept can make sure every image class share the same interface.

4. Polymorphism

This library uses the idea of Polymorphism to implement different operations. All operations inherit the base class `Operation`. It provides a few methods: (1) a `bool` randomizer to see if an operation take places this time; (2) a random number generator (between 0 and 1) to introduce randomness in each operation; and (3) a virtual function `perform` to ensure the same interface of its subclasses on image processing.

It looks like:

```
template<typename Image>
class Operation {
private:
    inline bool operate_this_time();
    inline _precision_type uniform_random_number();

public:
    virtual Image* perform(Image* image) = 0;
};
```

One subclass looks like

```
template<typename Image>
class ResizeOperation: public Operation<Image> {
public:
    Image* perform(Image* image) override;
};
```

A pipeline is made up of operations, and `Augmentor` stores the operations in a vector. We want to show the `Augmentor`'s ownership of operations. There are two safe ways to show ownership: (1) a vector of objects, (2) a vector of unique pointers. Since a vector of base classes casts the objects of subclasses into base-class objects, we decide to use `unique_ptr` to show the ownership.

```
// Augmentor.h
class Augmentor {
    std::vector<std::unique_ptr< Operation<Image> >> operations;
public:
    Augmentor& some_operation(parameters param...) {
        auto operation = std::make_unique<SomeOperation<Image>>(param);
        operations.push_back(std::move(operation));
        return *this;;
    }
}
```

The advantage for ownership lies in memory management. Before an `Augumentor` is going to be destroyed, the `unique_ptr` will first release the object it points to. Therefore, we can prevent any potential memory leaks in `Augmentor` class.

5. Compile-time programming

This library uses a lot of compile-time programming to optimize the code and performance. First of all, this library uses Template heavily. Since I have discussed Template in previous section, I will skip the basic usage of Template here. Instead, I will talk about how we can use Template to optimize code structure.

5.1. Different Implementations based on template parameters

5.1.1. Gaussian Blur Filter

Here I will show how to use template to make either static or dynamic filter when the `GaussianBlue` operation is built. The basic idea of blurring an image is to convolute between a filter and an image. The values of a filter can be either stored in array or vector. Storing in an array has advantages like fast accessing, but it requires users to specify the size before compile. Storing in a vector is more flexible, whose size is set in run-time, but it has relatively slow access. Therefore, I decide to use Template to integrate these two situations. The design looks like

```
template<unsigned N=0, bool Static = (N > 0) >
class gaussian_blur_filter_1D;

template<unsigned N>
class gaussian_blur_filter_1D<N, true> {
    double array[N];
public:
    explicit gaussian_blur_filter_1D(double sigma);
};

template<unsigned N>
class gaussian_blur_filter_1D<N, false> {
    std::vector<double> vector;
public:
    explicit gaussian_blur_filter_1D(double sigma, size_t n);
};
```

Either static and dynamic filters are created based on the constructor. Here is an example to build them.

```
// build a static filter with size of 5.
auto filter = gaussian_blur_filter_1D<5>(1.0);

// build a dynamic filter with size of 5.
auto filter = gaussian_blur_filter_1D(1.0, 5);
```

`BlurOperation` class wraps these two constructors into two member methods, so is the `Augmentor` class. Thus, users can build a blur operation either by `blur<5>(sigma)` or `blur(sigma, 5)`.

5.1.2. Random Number Generator

Here is another case to use template: allow a random number generator to output either integer or floating point numbers. In modern c++, `<random>` package is used to generate random numbers. It has two uniform number generators: `std::uniform_real_distribution` and `std::uniform_int_distribution`. Normally, if we want to build a generator like:

```
template <typename DataType>
class UniformDistributionGenerator<DataType> {
public:
    inline DataType operator()();
}
```

We may include both uniform number generators mentioned above as members, and call them based on datatype. However, we can slightly modify the template and split this template into two implementations:

```
template <typename DataType, bool IsReal = std::is_floating_point<DataType>::value>
class UniformDistributionGenerator;

class UniformDistributionGenerator<DataType, true> {
 std::uniform_real_distribution<DataType> distribution;
}

class UniformDistributionGenerator<DataType, true> {
 std::uniform_int_distribution<DataType> distribution;
}
```

In this way, there is no overhead from redundant members. Also, the performance is better since we don't have to do if-else evaluation when calling `operator()`. The performance difference here may not be significant due to the simple function here, but this idea can be applied to more complex design.

6. Features

There are other design features that distinguish our library from others.

6.1. More realistic randomness

Many image processing libraries (e.g. PIL) use `rand()` to generate random numbers, but this method may cause a few issues. please see this Q&A. On the contrary, our library uses `<random>` to achieve more realistic randomness. We use current timestamp as seed to create a generator, and then use `uniform_distribution` to output random numbers. Our library should result in better randomness than others.

6.2. Fast Gaussian Blur

This library implements some optimized algorithm to increase performance. One example is the Fast Gaussian Blur. Since Gaussian Blur is expensive, whose complexity should be at least O(N * r), where N is the area of an image and r is the size of a filter. However, research have found that multiple Box Blurs can approximate the result of Gaussian Blur, and the complexity of a Box Blur can be as low as O(N). Therefore, this library decides to implement the fast Gaussian Blur to increase the performance. The details can be found in the `Opperation.h` file.