

CS205 C/C++ Programming - Project5 Report

Name: 罗皓予 (Haoyu Luo)

SID: 12112517

Part 0 - Catalogue

Part 1 - Description

1. GitHub链接
2. 源码结构
3. 源码概述

Part 2 - Analysis

1. 模板矩阵类的分析与设计

- 1.1 模板类的编写与使用
- 1.2 矩阵类的分析与设计

2. 构造函数与析构函数

- 2.1 构造器
- 2.2 复制构造函数
- 2.3 析构函数

3. 运算符重载&友元函数

- 3.1 赋值运算符重载
- 3.2 访问元素运算符的重载
- 3.3 矩阵运算符重载
- 3.4 友元函数
- 3.5 矩阵的转置运算符重载
- 3.6 矩阵的相等运算符重载

4. ROI的分析与实现

4.1 ROI构建方法

5.矩阵类函数

5.1 在矩阵底部添加/删除若干行数据

5.2 矩阵通道的分离/合并

5.3 矩阵的合并（横向/纵向）

5.4 IO函数（文件存取和图像转换）

5.5 克隆，复制，重塑，转换

5.6 生成基本矩阵

Part 3 - Result & Verification

1.创建矩阵

2.复制构造函数以及赋值运算符重载的正确性验证

3.矩阵访问运算符重载的正确性实验

4.矩阵基本运算正确性检验

5.矩阵乘法正确性检验

6.ROI 相关操作正确性检验

7.在矩阵底部添加或删除元素的正确性检验

8.矩阵合并正确性验证

9.矩阵通道的分离/合并正确性验证

10.IO函数正确性验证

11.Hard Copy 与 Soft Copy 时间差异比较

Part 4 - Difficulties&Solutions

1.ROI设计

2.数据数组未初始化

Part 5 Conclusion

Part 1 - Description

1. GitHub链接

本次项目源码将存于Github仓库。

(<https://github.com/jimmylaw21/CS205-C-C-Program-Design/tree/main/project5>)

2. 源码结构

本次作业共上交2个文件:

1.main.cpp

2.Mat.hpp

3. 源码概述

3.1 关于main源码的解释说明

本次project的主要任务是设计一个矩阵类，该类应该包含矩阵的数据和行数、列数、通道数等相关信息。该类应该支持不同的数据类型。这意味着矩阵元素可以是unsigned char, short, int, float, double, 如果将矩阵对象分配给另一个对象，则不要使用内存硬拷贝。须仔细处理内存管理，以避免内存泄漏，并多次释放内存。同时要实现一些常用的操作符，包括但不限于=、==、+、-、*等，最好将project4中的矩阵乘法加速应用在这次project，以及实现感兴趣区域（ROI），以避免内存硬拷贝。

因此，本次main函数的主要功能就是进行试验，测试矩阵类各函数的正确性或性能表现。

3.2 关于Mat源码头文件的解释说明

0.学习链接:

[Leo-Adventure/CS205-CPP-projects: The repository contains all five projects implemented in Sustech CS205 C/C++ class \(github.com\)](#)

https://docs.opencv.org/master/d3/d63/classcv_1_1Mat.html

[confusion about Mat::data, datastart, dataend, datalimit, step · Issue #8304 · opencv/opencv \(github.com\)](#)

[OpenCV参考手册之Mat类详解（二） - Henry2017 - 博客园 \(cnblogs.com\)](#)

[纯C++代码实现将像素矩阵保存为bmp图片rs勿忘初心的博客-CSDN博客c++ 保存bmp](#)

[C++友元函数和友元类 \(C++ friend关键字\) \(biancheng.net\)](#)

进行了以下工作

1.构建了一个矩阵类，重写了构造函数与析构函数

2.实现了运算符重载&友元函数，包括+，-，乘号，/，=，+=，-=，乘等于，/=，！（转置），~（逆矩阵），++，--，==，!=，友元+，-，乘，/，<<，>>

3.用size_t rows，size_t cols，data指针，dataStart指针，dataEnd指针实现ROI

4.实现了一些矩阵类函数，包括文件存取和图像转换等等。

Part 2 - Analysis

1. 模板矩阵类的分析与设计

1.1 模板类的编写与使用

由于本次project中要求矩阵能够储存unsigned char, short, int, float, double等类型，同时对于不同类型而言，除了保存的对象类型不同之外，所有的操作的都是相同的。一种比较朴素的方法是编写多个功能相同，保存类型不同的矩阵类，然而这样会导致工作量巨大，而且代码会非常冗长，可维护性低，另一种比较朴素的方法是通过在头文件中使用typedef来为每一种类型创建不同的别名实现，然而这种实现在每次修改类型的时候都要编辑头文文件，不能让typedef同时代表两种不同的类型，因此并不能使用这种方法在同一个程序中同时定义不同类型的矩阵。

因此，本次project我会使用C++的模板类实现这一功能，我构建了一个模板矩阵类，将原本基于基本类型的矩阵类抽象成为模板类，对于模板矩阵类的成员函数，同样地使用模板成员函数替换

由课堂知识，我知道了模板并不是具体的类和成员函数，它们是C++编译器指令，只是说明了如何生成类和成员函数定义，所以与基础类的文件管理方式不同，由于模板并不是类和函数，所以不能将模板成员放在独立的cpp文件当中。在此处将所有模板信息都整合放在hpp文件当中。

对于对于模板类的使用，首先需要声明一个类型为模板类的对象，如

```
Mat<float> mat;
Mat<unsigned char> mat;
```

这种操作会使得编译器使用float类型或者unsigned char类型替换模板中所有的类型参数，因为与模板函数能够自动识别类型不同，模板类必须显式地提供所需的类型。

1.2 矩阵类的分析与设计

对本次project的要求进行分析，首先可以知道一个矩阵应该拥有描述该矩阵的行数row以及列数col以及通道数channel的成员变量，以及存储这个矩阵对应元素的元素数组data，。

由于对于一个矩阵而言一个元素可能有复数个通道，如三通道的RGB矩阵和四通道的RGB透明度矩阵，对于同一个元素而言，如果要进行不同通道值的访问，如果需要跳过一大段内存才能访问到属于同一个元素的不同通道的值，显然效率是非常低的，所以考虑到在应用当中的效率，对于同一个元素而言，不同通道的值应该存储在一块连续的内存当中。同时，由前两次project可知，复制矩阵时大段地复制内存是相当低效的，因此本次project仍使用data指针作为成员变量。

同时，对于同一块数组区域，可能会有多个矩阵共用这一块区域，尤其是为了节省内存和时间以及进行ROI操作时我们会使用软拷贝，所以可以使用一个指针ref_count记录下当前共同使用这一块内存区域的矩阵对象个数，防止不同指针多次释放同一块内存导致错误。

由于矩阵的元素是通过使用一维数组进行存储，所以需要维护一个跨度变量span，指示跨行访问元素需要跨过的元素个数，这个跨度变量的默认值设置为列数乘以通道数，但是对于子矩阵而言，这个跨度变量应该设置为母矩阵的列数乘以通道数，因为子矩阵使用的是母矩阵的元素数组的内存区域。

然后，为了实现ROI操作，我学习了OPENCV的Mat类的设计，使dataStart指针，dataEnd指针成为矩阵成员变量，他们分别指向母矩阵的第一个元素和母矩阵的最后一个元素的下一个元素,data指向的是ROI矩阵的第一个元素, 那么我们的data指针实际上指向的是ROI矩阵的第一个元素，即每一个矩阵实际上都描述着一个ROI矩阵，那么data+ rows * channels * cols指向的是ROI矩阵的最后一个元素的下一个元素。

这里，我们需要注意的是跨行访问的跨度变量的默认值设置为列数乘以通道数，但是对于子矩阵而言，这个跨度变量应该设置为母矩阵的列数乘以通道数，因为子矩阵使用的是母矩阵的元素数组的内存区域。所以我们要维护一个跨度变量span，它是矩阵的成员变量，每次生成ROI矩阵时，子矩阵应当继承母矩阵的span，dataStart，dataEnd。

此外，矩阵类还应该有一个静态变量matrices_num用于记录矩阵类对象个数，便于我们进行内存管理和debug。

最后，我们应当注意到矩阵各个参数都应该是正整数或零，为了防止负数导致的错误，这些参数都应该使用size_t类型。

以下是Mat类的成员变量。

```
template <class T>
class Mat {
private:
    size_t rows;
    size_t cols;
    size_t channels;
    size_t span;
    // dataStart指向的是矩阵的第一个元素，
    // dataEnd指向的是矩阵的最后一个元素的下一个元素，
    // data指向的是ROI矩阵的第一个元素，
    // data+ rows * channels * cols指向的是ROI矩阵的最后一个元素的下一个元素
    T* data;
    T* dataEnd;
    T* dataStart;
    static size_t num_matrices;
    size_t* ref_count;
```

2. 构造函数与析构函数

2.1 构造器

通过使用默认参数的方式，定义默认构造函数，对所有成员进行初始化，确保所有成员一开始就有已知的合理的值。同样地，在默认构造函数当中也需要对传入的参数进行合法检查，确保没有空指针等问题。以及ROI区域被母矩阵（dataStart，dataEnd）包含，不会越界。然后根据矩阵所包含元素的个数对元素数组进行内存的分配，最后给表示当前存在矩阵个数的静态变量num_matrices 值加一。

为了方便用户的使用，我设计了两个默认构造函数，其中一个只需要三个参数，剩余参数将会进行默认复制。

```
this->rows = rows;
this->cols = cols;
this->channels = channels;
this->data = new T[rows * cols * channels];
//data数组全置0
memset(this->data, 0, rows * cols * channels * sizeof(T));
this->dataEnd = data + rows * cols * channels;
this->dataStart = data;
this->span = cols * channels;
this->ref_count = new size_t(1);
```

方法如下：

```
template <typename T>
Mat<T>::Mat(size_t rows, size_t cols, size_t channels);
template <typename T>
Mat<T>::Mat(size_t rows, size_t cols, size_t channels, size_t span, T* data =
nullptr, T* dataEnd = nullptr, T* dataStart = nullptr, size_t* ref_count =
nullptr);
```

之后便可以使用默认构造器声明对象变量：

```
Matrix mat;
Matrix mat = Matrix();
Matrix *mat = new Matrix();
```

但是，值得注意的是，由于对于元素数组传入的变量仅仅是一个data指针，所以是有可能存在将同一块数组空间同时传递给多个矩阵的情况，但是由于不通过矩阵来传递data指针，不同的矩阵并不能检测到自己的元素数组空间是否被多个矩阵对象共用，这样会使得对一个矩阵进行的操作可能会影响到另一个矩阵的数据安全问题，所以在默认构造函数内部应当使用memcpy方法，高效且安全地将传入数组的值复制一遍作为自己的元素数组，避免隐式地影响多个矩阵data数组的值。

另一方面，由于在实际应用中，我们常常在传入大规模的矩阵时需要使用文件进行元素数组的赋值，所以对于矩阵对象构造器还需要重载一个根据文件流来创建矩阵的方法。

```
template <typename T>
Mat<T>::Mat(size_t row, size_t col, size_t channel, ifstream& file);
```

该构造函数可以成功根据传入的文件流创建对应的元素数组。

2.2 复制构造函数

由课程知识，我知道现如果没有显示地重载一个类的复制构造器，编译器便会自动地生成复制构造函数，但是这种默认生成的复制构造函数只会对对象中非静态的成员进行软拷贝，对于指针变量来说，即直接将地址复制给该对象，这将导致只要两个对象当中的一个被释放，该地址的内存就将会被释放，而此时原始对象的成员指针仍然指向该内存块，这有可能导致同一块内存被释放两次，从而使程序崩溃。因此，我们需要重定义复制构造函数。

在复制构造函数当中，传入的变量是一个同类的对象变量的引用，因为使用按值传递的方法将耗费一定的时间和内存去将传入的对象复制给一个临时变量，而使用引用传递，可以避免大段内存的复制，使用的是原矩阵地数据而非副本。

所以该函数需要对新矩阵变量进行初始化，对于矩阵的元素数组，如果根据原矩阵的数据对里面的值进行硬拷贝，则会导致效率低下，尤其是当矩阵元素数组非常大的时候。所以考虑将新矩阵的元素数组指针直接指向原矩阵元素数组的指针指向的内存，为了避免同时释放一块内存区域两次的情况，在将新矩阵元素数组的指针指向原矩阵元素数组的指针指向的内存的同时，将表示引用次数的指针ref_count代表的值加一，在释放矩阵的时候需要检查该值是否为1，如果大于1则不执行对该元素数组指针指向的内存区域进行释放，这样既避免了大段内存的复制，又安全地避免了同一块内存不同矩阵的data指针被释放两次的情况。

```

template <typename T>
Mat<T>::Mat(const Mat& other) : rows(other.rows), cols(other.cols),
channels(other.channels), span(other.span),
data(other.data), dataEnd(other.dataEnd), dataStart(other.dataStart),
ref_count(other.ref_count){
    num_matrices++;
    (*ref_count)++;
}

```

2.3 析构函数

矩阵类的析构函数的作用是对矩阵的元素数组指向的内存进行释放，避免内存泄漏。由于为元素数组分配内存时使用的是new[]分配函数来开辟数组空间而非单变量空间，所以在释放该空间的时候也要使用delete[]释放函数来释放这一数组空间。

由于一个矩阵对象的元素数组区域有可能同时被多个矩阵对象的元素数组所指向，所以在析构函数当中需要首先使当前内存区域引用次数的值*ref_count减一，然后判断其是否为零，如果为零，则说明该区域没有矩阵对象在使用，此时应该直接释放该区域，否则不对该区域进行释放，最后将表示当前存在矩阵类对象个数的变量值matrices_num减一。

```

template <typename T>
Mat<T>::~~Mat(){
    if(--(*ref_count) == 0){
        delete[] data;
        this -> data = nullptr;
        delete[] ref_count;
        this -> ref_count = nullptr;
        num_matrices--;
    }
}

```

3. 运算符重载&友元函数

3.1 赋值运算符重载

在创建了两个对象之后，如果想要对其中一个对象赋值为另一个同类型的对象，我们希望直接使用=进行赋值操作，在没有重载赋值运算符的情况下，编译器会自动地生成对复制运算符的重载，执行着默认的复制构造函数的功能，只执行对赋值源对象的非静态成员变量的软拷贝，将成员指针变量代表的地址复制给待复制对象，所以也会因为同一块内存空间被释放了两次而使程序崩溃。因此，我们需要重载赋值运算符。

首先需要进行程序安全性的检查，然后为了防止自身给自身赋值带来不必要的开销，在真正开始赋值之前先判断两个对象是否完全相同，如果完全相同，则返回原对象，如果不相同，则进行析构函数的操作，并将自身指向元素数组的指针指向赋值矩阵对象的元素数组空间，并将引用该空间的值加一。如果不进行对原有数组空间内存的释放，在对其进行新的赋值之后，原先指向的内存区域将在程序运行的时候无法释放，可能会造成严重的内存泄漏甚至导致程序因空间不足而终止。最终返回待赋值对象的引用。

```

if (this != &other){
    if(--(*ref_count) == 0){
        delete[] data;
        this -> data = nullptr;
        delete[] ref_count;
    }
}

```



```

        this->ref_count = nullptr;
        num_matrices--;
    }
    this->rows = other.rows;
    this->cols = other.cols;
    this->channels = other.channels;
    this->data = other.data;
    this->dataEnd = other.dataEnd;
    this->dataStart = other.dataStart;
    this->span = other.span;
    this->ref_count = other.ref_count;
    (*ref_count)++;
}
return *this;

```

3.2 访问元素运算符的重载

为了矩阵元素以及其通道元素访问的便捷性和代码的简洁性，我重载了()运算符和[]运算符，通过()当中传入代表矩阵行数以及列数以及通道数的数字，可以直接访问到该坐标存储的元素值，或者通过[]当中传入代表矩阵行数以及列数，可以直接访问到该坐标存储的元素值中第一个通道的值，或者通过[]当中传入index，以直接访问到该坐标存储的元素值，由于在矩阵类当中，同一个元素的不同通道的数值是存储在一起的，即存储在一维数组相邻的位置上，所以对于输入的行数r，列数c，通道数ch，返回对应元素的计算公式为：

```

return data[row * span + col * channels + channel];
return data[row * span + col * channels];
return data[index];

```

其中row * span代表在二维数组中所需要跳过的行数中所包含的元素数量，col * channel代表在二维数组当中一行存储的元素个数为列数乘以通道数，对于输入的目标行数r，则要先跳过r之前的所有元素来到目标的行，再通过输入的列数c，跳过在当前行之中前面的c个列中代表的元素，最后根据输入的通道数获取相应的元素并返回。

通过实现矩阵元素访问运算符的重载，可以在之后的操作当中更方便地访问矩阵元素，由于返回值是引用类型，所以也可以通过这个运算符实现对矩阵元素值的修改。

3.3 矩阵运算符重载

首先介绍的是本矩阵类对于标量的运算符重载。以下为各方法。

```

Mat& operator+=(const T& value);
Mat& operator-=(const T& value);
Mat& operator*=(const T& value);
Mat& operator/=(const T& value);

Mat operator+(const T& value) const;
Mat operator-(const T& value) const;
Mat operator*(const T& value) const;
Mat operator/(const T& value) const;

```

以矩阵的加法为例，对于+运算符的重载，不能改变传入的矩阵以及主调矩阵成员元素的值，所以函数头需要将两个变量的使用都设置为const，而对于+=运算符的重载，由于需要改变主调矩阵成员元素的值，因此只对传入的矩阵设置为const。

其次，+运算符要求返回一个全新的矩阵，其中的元素为源矩阵元素与传入标量相加的结果，所以需要在函数内部创建一个新的矩阵对象并进行对其元素数组的赋值操作，由于改变了元素数组的值，所以无法与两个被加数组进行空间的共用。由于重载了矩阵的访问运算符，所以对于矩阵的加法，只需要逐行逐列逐通道地对矩阵元素进行逐个相加即可。

但是，在本矩阵类中，+=运算符先被重载，它要求返回一个主调矩阵的引用，其余部分与+运算符操作相似，即则不在函数当中创建新的对象，直接在*this 代表的对象的数组元素当中进行逐个元素的相加，因此+=运算符的设计即为创建一个新的矩阵对象，然后对其进行+=运算符运算。

此外，值得注意的是该矩阵对象拥有的元素数组空间有可能同时被多个矩阵对象所共用，对这一内存区域值的修改会同时改变多个矩阵的元素值，这会导致不必要的bug和数据篡改，所以在进行+=运算符重载的时候需要判断该空间是否只有它在使用，如果是，则可以直接进行修改，否则需要将原有的引用指针代表的值减一，代表原矩阵已经不再使用原来的元素数组空间了，然后重新开辟一个新的空间，并将新的值赋值给新的元素数组，然后将代表引用自身数组空间的值设置为1。

+=运算符：

```
size_t size = rows * cols * channels;
if (*ref_count == 1){
    for (size_t i = 0; i < size; i++){
        data[i] += value;
    }
}
else{
    T* temp = new T[size];
    for (size_t i = 0; i < size; i++){
        temp[i] = data[i] + value;
    }
    data = temp;
    dataEnd = data + size;
    dataStart = data;
    (*ref_count)--;
    ref_count = new size_t(1);
}
return *this;
```

+运算符：

```
template <typename T>
Mat<T> Mat<T>::operator+(const Mat<T>& other) const{
    Mat<T> result(*this);
    result += other;
    return result;
}
```

那么，矩阵与标量的运算符重载就完成了，相对比较简单。

进一步地，我们需要对矩阵和矩阵地运算符进行重载。以下为各方法。

```

Mat& operator+=(const Mat& other);
Mat& operator-=(const Mat& other);
Mat& operator*=(const Mat& other); //dot
Mat& operator/=(const Mat& other); //dot

Mat operator+(const Mat& other) const;
Mat operator-(const Mat& other) const;
Mat operator*(const Mat& other) const;
Mat operator/(const Mat& other) const;

```

其中需要特别说明的是，由于 $*$ 和运算符 $/$ 运算符涉及到不同大小的内存空间转换，为避免出现内存管理错误，它们所代表的计算为点计算，而非叉乘和求逆，但乘法运算符和 $/$ 运算符执行了这些运算功能。

其中， $+$ ， $+=$ ， $-$ ， $-=$ ， $*$ ， $/$ 的运算与上述矩阵与标量计算别无二致，区别在于主调矩阵每一个元素不再是与固定值运算，而是与传入矩阵对应位置的元素进行计算。

对于乘法运算符，有了矩阵元素访问运算符的重载，矩阵乘法的表示也更加地直观，通过重载 $*$ 运算符，可以实现矩阵的直接叉乘。

在该程序中，我们首先创建一个新的大小符合矩阵叉乘结果的矩阵，用其装载矩阵叉乘的结果，避免了在原内存空间上的改动，对于多通道矩阵的初步乘法实现，可以依次对不同通道的元素先进行运算，对于每一个通道的矩阵元素，都分别使用传统的矩阵乘法，同样可以使用O3编译优化，openMP并行加速和访存优化进行矩阵乘法的加速，这里使用了kij的运算顺序对矩阵进行相乘，在最外层的循环实现不同通道的切换，这样就可以实现初步的矩阵叉乘。

```

#pragma omp parallel for
    for (size_t k = 0; k < other.rows; k++){
        for (size_t i = 0; i < rows; i++){
            for (size_t j = 0; j < other.cols; j++){
                for (size_t ch = 0; ch < channels; ch++){
                    result(i, j, ch) += (*this)(i, k, ch) * other(k, j, ch);
                }
            }
        }
    }
}

```

对于除法运算符，有了矩阵元素访问运算符的重载，矩阵除法的表示也更加地直观，通过重载 $/$ 运算符，可以实现矩阵的求逆，然后叉乘。

在该程序中，我们首先创建一个新的大小符合矩阵叉乘结果的矩阵，用其装载矩阵叉乘的结果，避免了在原内存空间上的改动，然后对传入矩阵进行求逆操作，具体方法后续部分会讲解，最后对主调矩阵和传入矩阵的逆矩阵进行叉乘操作获除法结果。

```

Mat<T> other_inv(other.rows, other.cols, other.channels);
this -> inv(other_inv);
#pragma omp parallel for
    for (size_t k = 0; k < other_inv.rows; k++){
        for (size_t i = 0; i < rows; i++){
            for (size_t j = 0; j < other_inv.cols; j++){
                for (size_t ch = 0; ch < channels; ch++){
                    result(i, j, ch) += (*this)(i, k, ch) * other_inv(k, j, ch);
                }
            }
        }
    }
}

```

```

    }
}

return result;

```

此外，本project还重载了自增运算符和自减运算符，包括左值和右值为矩阵对象两种情况，若矩阵自增，则每个元素的每个通道值加一，若矩阵自减，则每个元素的每个通道值减一。

```

Mat& operator++();
Mat& operator--();
Mat operator++(int);
Mat operator--(int);

```

但是重载运算符只能解决左值为矩阵对象，右值为运算数的情况，对于顺序反过来之后的情况就无法再适配，这时候可以使用友元函数帮助解决这个问题。

3.4 友元函数

在C++中，一个类中可以有 public、protected、private 三种属性的成员，通过对象可以访问 public 成员，只有本类中的函数可以访问本类的 private 成员。但是，存在一种例外情况——友元（friend），借助友元（friend），可以使得其他类中的成员函数以及全局范围内的函数访问当前类的 private 成员。

友元函数是在当前类以外定义的、不属于当前类的函数，也可以在类中声明，但要在前面加 friend 关键字，这样就构成了友元函数。友元函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数。友元函数可以访问当前类中的所有成员，包括 public、protected、private 属性的。

在友元函数原型声明时，使用friend关键字，将非本类的想作为其第一个操作数，就可以通过友元函数实现操作数顺序的反转。对于+、-、*、/运算符的重载，为了适配首项不是目标类的运算情况，都统一地使用了友元函数实现。

```

// friend operators
template<class TT>
friend Mat operator+(TT num, const Mat<TT>& mat){
    return mat + num;
}
template<class TT>
friend Mat operator-(TT num, const Mat<TT>& mat){
    return mat - num;
}
template<class TT>
friend Mat operator*(TT num, const Mat<TT>& mat){
    return mat * num;
}
template<class TT>
friend Mat operator/(TT num, const Mat<TT>& mat){
    return mat / num;
}

```

这里，+、-、*、/的友元函数运算符重载调用了之前的运算符重载，比较简洁地执行了相似的功能。

在矩阵类的实际使用和开发调试中，我们经常需要输出矩阵来查看我们操作的效果，如果使用传统的方法，则需要每一次查看都要写一遍遍历所有元素的函数，再重复调用，会使得代码冗长，开发效率低下。因此，本次project重载了<<, >>运算符，方便直观且快捷地输出矩阵，使之能够与cout一起显示矩阵的每一个元素以及通道的值，以及输入矩阵，便于IO操作输入矩阵。

```
template<class TT>
friend ostream& operator<<(ostream& os, const Mat<TT>& mat)
{
    size_t r = mat.rows;
    size_t c = mat.cols;
    size_t ch = mat.channels;
    for(size_t i = 0; i < r; i++){
        for(size_t j = 0; j < c; j++){
            os << "{";
            for(size_t k = 0; k < ch; k++){
                os << mat(i, j, k);
                if(k != ch - 1){
                    os << " ";
                }
            }
            os << "}";
            if(j != c - 1){
                os << ", ";
            }
        }
        if(i != r - 1){
            os << "\n";
        }
    }
    return os;
}

template<class TT>
friend ostream& operator>>(ostream& os, const Mat<TT>& mat){
    for(size_t i = 0; i < mat.rows; i++){
        for(size_t j = 0; j < mat.cols; j++){
            for(size_t k = 0; k < mat.channels; k++){
                os >> mat.data[i * mat.span + j * mat.channels + k];
            }
        }
    }
    return os;
}
```

当需要逐个显示数组当中的每一个元素以及其拥有的所有通道的值时，需要对每一行的元素逐个输出，对于一行当中的每一个元素而言，其拥有的是不同的通道数所代表的值，因此对于每一行元素又需要在遍历元素的过程当中遍历该元素通道的所有值。同时用花括号将同一元素的通道值括起来，便于直观地检查结果。输出样例如下：

```

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = (i + 1) * (j + 1);
        mat3(i, j, 1) = (i + 1) * (j + 1) + 1;
        mat3(i, j, 2) = (i + 1) * (j + 1) + 2;
    }
}
{1 2 3}, {2 3 4}, {3 4 5}
{2 3 4}, {4 5 6}, {6 7 8}
{3 4 5}, {6 7 8}, {9 10 11}

```

3.5 矩阵的转置/求逆运算符重载

在学习了OpenCV的Mat类一些实现方法，发现在实际应用当中对于矩阵的转秩运算和求逆运算也十分重要，所以在本程序当中也使用!运算符来实现对矩阵的转秩操作和~运算符来实现对矩阵的求逆操作。

通过使用()矩阵访问运算符，可以简洁直观地对矩阵进行转秩操作。对于!运算符的重载，返回的是一个副本矩阵，所以在函数中新创建了一个新的矩阵，该矩阵的所有成员变量都和源矩阵相同，之后对它的元素数组进行转置赋值，!运算符调用了transpose方法，该方法核心代码如下：

```

size_t size = rows * cols * channels;
T* temp = new T[size];
for (size_t i = 0; i < rows; i++){
    for (size_t j = 0; j < cols; j++){
        for (size_t ch = 0; ch < channels; ch++){
            {
                temp[j * rows * channels + i * channels + ch] = (*this)(i, j,
ch);
            }
        }
    }
}
dst = Mat<T>(rows, cols, channels, temp);
return dst;

```

通过使用()矩阵访问运算符，可以简洁直观地对矩阵进行转秩操作。对于~运算符的重载，返回的是一个副本矩阵，所以在函数中新创建了一个新的矩阵，该矩阵的所有成员变量都和源矩阵相同，之后对它的元素数组利用行列式值进行代数余子式方法求逆赋值，~运算符调用了inv方法，该方法核心代码如下：

```

size_t size = rows * cols * channels;
T* temp = new T[size];
for (size_t i = 0; i < rows; i++){
    for (size_t j = 0; j < cols; j++){
        Mat<T> temp(rows - 1, cols - 1, channels);
        temp.dataEnd = temp.data + (rows - 1) * (cols - 1) * channels;
        temp.dataStart = temp.data;
        for(int k = 0; k < rows - 1; k++){
            for(int l = 0; l < cols - 1; l++){
                for(int ch = 0; ch < channels; ch++){
                    if(k < i){

```

```

        if(l < j){
            temp(k, l, ch) = (*this)(k, l, ch);
        }else{
            temp(k, l, ch) = (*this)(k, l + 1, ch);
        }
    }else{
        if(l < j){
            temp(k, l, ch) = (*this)(k + 1, l, ch);
        }else{
            temp(k, l, ch) = (*this)(k + 1, l + 1, ch);
        }
    }
}
}
}
temp(j, i, 0) = pow(-1, i + j) * temp.det();
}
}
//复制temp到data
memcpy(dst.data, temp, size * sizeof(T));
delete[] temp;
return dst;

```

而求逆函数又利用了矩阵的行列式值，因此在inv函数中又调用了det函数。其核心代码如下：

```

T result = 0;
if(rows == 1){
    result = (*this)(0, 0, 0);
}else if(rows == 2){
    result = (*this)(0, 0, 0) * (*this)(1, 1, 0) - (*this)(0, 1, 0) *
(*this)(1, 0, 0);
}else{
    for(int i = 0; i < rows; i++){
        Mat<T> temp(rows - 1, cols - 1, channels);
        temp.dataEnd = temp.data + (rows - 1) * (cols - 1) * channels;
        temp.dataStart = temp.data;
        for(int j = 0; j < rows - 1; j++){
            for(int k = 0; k < cols - 1; k++){
                for(int ch = 0; ch < channels; ch++){
                    if(k < i){
                        temp(j, k, ch) = (*this)(j + 1, k, ch);
                    }else{
                        temp(j, k, ch) = (*this)(j + 1, k + 1, ch);
                    }
                }
            }
        }
        result += pow(-1, i) * (*this)(0, i, 0) * temp.det();
    }
}

return result;

```

3.6 矩阵的相等运算符重载

本次project中，还实现了矩阵==运算符，!=运算符的重载，==运算符会判断两个矩阵的每一个元素是否一一相等。

```
template <typename T>
bool Mat<T>::operator==(const Mat<T>& other) const{
    if (rows != other.rows || cols != other.cols || channels != other.channels){
        return false;
    }
    size_t size = rows * cols * channels;
    for (size_t i = 0; i < size; i++){
        if (data[i] != other.data[i]){
            return false;
        }
    }
    return true;
}
template <typename T>
bool Mat<T>::operator!=(const Mat<T>& other) const{
    return !(*this == other);
}
```

4.ROI的分析与实现

4.1 ROI构建方法

对于图像处理方面，ROI 代表一幅图像中感兴趣区域的信息，在矩阵当中，ROI 区域可以看作是一个矩阵的子矩阵，可以在矩阵当中维护相关的ROI 信息，在每次需要提取ROI 的时候只需要根据矩阵中的相关信息返回子矩阵即可。

我们可以通过硬拷贝的方式生成子矩阵，直接通过在函数当中直接创建一个新的矩阵对象进行返回，对于返回的ROI子矩阵，它的元素数组依次根据源矩阵的ROI区域的范围进行填充。

但是这样操作的效率在ROI区域较大的时候由于需要拷贝大段的内存会导致效率非常低下，而且很多情况下子矩阵不需要脱离母矩阵进行分析和操作，所以考虑使用返回子矩阵引用的方式实现ROI操作。

下面是在母矩阵当中子矩阵元素的提取公式和代码：

$$rows * srcColumn * srcChannel + cols * srcChannel + channels$$

由于在本矩阵类的设计中，子矩阵只会保留最原始母矩阵，即矩阵内存空间第一个元素位置的指针和最后一个元素下一个位置的指针，没有保存srcColumn，因此该任务由span承担，所以子矩阵的span和channel直接继承母矩阵的相应参数，所以实际上提取公式如下：

$$rows * span + cols * channel + channels$$

同时，提取ROI矩阵，用户只需输入row, col, height, width四个值即可，分别代表了子矩阵第一个元素地址和母矩阵第一个元素地址的长宽偏移值，以及子矩阵的rows和cols。

```
template <typename T>
Mat<T> Mat<T>::ROI(size_t row, size_t col, size_t height, size_t width) const;
```

以下是ROI构建方法中的核心代码：


```

dst.rows = height;
dst.cols = width;
dst.channels = channels;
dst.span = cols * channels;
dst.data = data + row * span + col * channels;
dst.dataStart = dataStart;
dst.dataEnd = dataEnd;
dst.ref_count = ref_count;
(*ref_count)++;
num_matrices++;

```

5.矩阵类函数

5.1 在矩阵底部添加/删除若干行数据

在参考OpenCV方法的时候发现有在矩阵底部添加和删除若干行数据的函数，于是思考对这两个方法的实现。

对于添加操作，通过传入希望在原矩阵添加的数组指针以及描述该数组行数以及列数以及通道数的信息，可以进行在原矩阵底部添加传入的数组。在对传入的参数进行检查后，创建一个大小为原矩阵对象元素数组大小加上传入数组大小的新数组，将原元素数组的元素——赋值到新数组前半部分，将新传入数组指针的元素——赋值到新数组后半部分。在赋值完成之后依然需要判断原矩阵的内存空间被多少个矩阵同时共享，将原有的引用值减一，如果没有矩阵使用则释放原有的空间，否则并创建一个新的引用值并将该值设置为1，之后将数组指针指向新创建数组的内存空间，这一操作就相当于完成了在原矩阵底部添加了传入的数组。核心代码如下：

```

int src_size = rows * cols * channels;
int tar_size = row * col * ch;
T * newdata = new T[src_size + tar_size];

memcpy(newdata, data, src_size * sizeof(T));
memcpy(newdata + src_size, data, tar_size * sizeof(T));
this->ref_count[0]--;

if(this->ref_count[0] == 0){
    delete[] this->data;
}else{
    this->ref_count = new int[1];
    this->ref_count[0] = 1;
}

this->data = newdata;
this->rows = this->rows + row;
return *this;

```

对于删除操作，只需要传入希望删除数据的行数即可。在检查数据是否合法后，在矩阵底部添加若干行数据同理，依然可以在函数内部创建一个大小为原矩阵元素数组大小减去希望删除数组大小的新元素数组，然后依次将原元素数组的值赋值到新矩阵当中，之后为了防止内存泄漏，先释放原元素数组的空间，再将原矩阵对象的数组指针指向新的数组代表的内存区域，最后返回对应的矩阵即可。

```

int tar_size = (rows - row) * cols * channels;
T * newdata = new T[tar_size];

```

```

memcpy(newdata, data, tar_size * sizeof(T));
this->ref_count[0]--;

if(this->ref_count[0] == 0){
    delete[] this->data;
}else{
    this->ref_count = new int[1];
    this->ref_count[0] = 1;
}

this->data = newdata;
this->rows = this->rows - row;
return *this;

```

5.2 矩阵通道的分离/合并

在参考OpenCV实现方法的过程中发现有对矩阵通道进行分离和合并的方法，在实际的应用过程当中矩阵合并可以被应用于单独处理每一个通道层的信息。

对于分离操作，我们需要将多通道矩阵分离为多个单通道矩阵，因此传入参数为一个预先准备好的vector<Mat>容器，用于装载分离出的多个单通道矩阵。在进行分离操作前，我们需要判断矩阵的通道数是否为一，若为一，则直接将该矩阵装入vector容器中返回，函数终止，若不为一，则对于每个通道单独生成一个单通道矩阵，然后遍历原矩阵对新矩阵们进行赋值，并将新矩阵装入vector容器中返回。

```

//将多通道矩阵分离为多个单通道矩阵
if (channels == 1){
    mv.push_back(*this);
    return;
}
for (size_t i = 0; i < channels; i++){
    Mat<T> mat(rows, cols, 1);
    for (size_t j = 0; j < rows; j++){
        for (size_t k = 0; k < cols; k++){
            mat.data[j * cols + k] = data[j * cols * channels + k * channels
+ i];
        }
    }
    mv.push_back(mat);
}

```

对于合并操作，我们需要将多个单通道矩阵分离为多通道矩阵，因此传入参数为一个预先准备好的装有单通道矩阵的vector<Mat>容器，用于提取多个单通道矩阵。在进行合并操作前，我们需要判断vector容器的大小是否为一，若为一，则直接返回这唯一的单通道矩阵，函数终止，若不为一，则在进行数据合法性检测后，生成一个多通道矩阵，通道数等于vector容器的大小，然后遍历vector容器里的原矩阵对新矩阵进行赋值，最后返回新矩阵。

```

if (mv.size() == 1){
    *this = mv[0];
    return *this;
}
size_t rows = mv[0].rows;
size_t cols = mv[0].cols;
size_t channels = mv.size();
for (size_t i = 0; i < mv.size(); i++){

```

```

        if (mv[i].rows != rows || mv[i].cols != cols){
            std::cerr << "In function merge..." << std::endl;
            std::cerr << "The input matrix should have the same size." <<
std::endl;
            throw "The input matrix should have the same size";
        }
    }
    Mat<T> mat(rows, cols, channels);
    for (size_t i = 0; i < rows; i++){
        for (size_t j = 0; j < cols; j++){
            for (size_t k = 0; k < channels; k++){
                mat.data[i * cols * channels + j * channels + k] = mv[k].data[i
* cols + j];
            }
        }
    }
    *this = mat;
    return *this;

```

5.3 矩阵的合并（横向/纵向）

在参考OpenCV实现方法的过程中发现有对矩阵进行合并的方法，在实际的应用过程当中矩阵合并可以被应用于图形的拼接当中。

对于矩阵的纵向合并，在确认传入参数合法之后，在函数内部创建一个新的矩阵，将这个矩阵的行数赋值为两个矩阵的行数之和，列数和通道数与两个矩阵保持一致，而其余参数为构造函数设定的默认值即可。对于新矩阵元素数组的赋值，在纵向合并的情况当中依然是先后将其赋值为源矩阵以及传入矩阵对应的元素数组的值，最后返回新创建的矩阵对象就能完成矩阵的纵向合并。

```

Mat<T> new_mat(rows + mat.rows, cols, channels);
//将原矩阵的数据复制到新矩阵中
memcpy(new_mat.data, data, rows * cols * channels * sizeof(T));
//将输入矩阵的数据复制到新矩阵中
memcpy(new_mat.data + rows * cols * channels, mat.data, mat.rows * mat.cols
* mat.channels * sizeof(T));
return new_mat;

```

对于矩阵的横向合并，在确认传入参数合法之后，实现方式与纵向合并相似，需要获取两个矩阵每一行对应的元素的值，再分别对其进行赋值。

```

Mat<T> new_mat(rows, cols + mat.cols, channels);
//将原矩阵的数据复制到新矩阵中
for (size_t i = 0; i < rows; i++){
    memcpy(new_mat.data + i * new_mat.cols * channels, data + i * cols *
channels, cols * channels * sizeof(T));
}
//将输入矩阵的数据复制到新矩阵中
for (size_t i = 0; i < mat.rows; i++){
    memcpy(new_mat.data + i * new_mat.cols * channels + cols * channels,
mat.data + i * mat.cols * mat.channels, mat.cols * mat.channels * sizeof(T));
}
return new_mat;

```

5.4 IO函数（文件存取和图像转换）

本次project中，还针对矩阵的实际使用准备了一些IO函数和图像函数，方便用户存储和使用矩阵，以及通过矩阵生成图像和将图像转换成矩阵。函数如下：

```
void save(const string& filename) const;
void load(const string& filename);
void saveAsBMP(const std::string& filename) const;
void loadFromBMP(const std::string& filename);
Mat rgbtorgay() const;
```

对于save方法，传入参数为用户自定义的文件名，方法会将矩阵的基本参数和data元素以空格间隔写入位于本地的以该文件名命名的txt文件，具体实现如下：

```
template <typename T>
void Mat<T>::save(const std::string& filename) const{
    std::ofstream ofs(filename + ".txt");
    if (!ofs){
        std::cerr << "In save function..." << std::endl;
        std::cerr << "Can't open the file." << std::endl;
        exit(EXIT_FAILURE);
    }
    ofs << rows << " " << cols << " " << channels << std::endl;
    size_t size = rows * cols * channels;
    for (size_t i = 0; i < size; i++){
        ofs << data[i] << " ";
    }
    ofs.close();
}
```

对于load方法，传入参数为用户需要读入数据的文件名，方法会读取文件中的矩阵基本参数信息和元素数组信息来生成矩阵，其余参数为默认构造函数的默认值，具体实现如下：

```
template <typename T>
void Mat<T>::load(const std::string& filename){
    std::ifstream ifs(filename + ".txt");
    if (!ifs){
        std::cerr << "In load function..." << std::endl;
        std::cerr << "Can't open the file." << std::endl;
        exit(EXIT_FAILURE);
    }
    ifs >> rows >> cols >> channels;
    size_t size = rows * cols * channels;
    data = new T[size];
    dataStart = data;
    dataEnd = data + size;
    ref_count = new size_t(1);
    for (size_t i = 0; i < size; i++){
        ifs >> data[i];
    }
    ifs.close();
}
```

对于saveAsBMP方法，方法会将三通道矩阵转换为RGB图像，保存为bmp格式。首先，它检查 Mat 类型中存储的图像的通道数是否为 3，因为 BMP 图像的每个像素都由 3 个 8 位的通道组成（分别为红、绿、蓝）。如果不是 3，则在 stderr 中输出错误信息并退出程序。

然后，它还检查 Mat 类型中存储的图像的每个像素的通道值是否在 [-128, 127] 范围内。BMP 图像的通道值应该在 [0, 255] 范围内，如果不是，则在 stderr 中输出错误信息并退出程序。

接下来，它打开一个文件输出流，并向文件写入 BMP 图像的文件头和信息头。文件头是一个 14 字节的数组，其中包含了 BMP 图像的文件类型、文件大小等信息。信息头是一个 40 字节的数组，其中包含了 BMP 图像的图像尺寸、图像格式、图像分辨率等信息。

然后，它还在文件中写入 BMP 图像的数据。它先创建一个 char 类型的数组来存储转换后的图像数据，然后按行扫描 Mat 类型中存储的图像数据，并将其复制到新创建的数组中。每个像素由 3 个 8 位通道组成，因此将每个像素的每个通道分别复制到新数组中。

最后，将新数组中的数据写入文件，然后关闭文件输出流并释放内存。

```
template <typename T>
void Mat<T>::saveAsBMP(const std::string& filename) const{
    if (channels != 3){
        std::cerr << "In saveAsBMP function..." << std::endl;
        std::cerr << "The channels of the image should be 3." << std::endl;
        exit(EXIT_FAILURE);
    }
    size_t size = rows * cols * channels;
    for (size_t i = 0; i < size; i++){
        if (data[i] < -128 || data[i] > 127){
            std::cerr << "In saveAsBMP function..." << std::endl;
            std::cerr << "The value of each channel should be in [0, 255]." <<
std::endl;
            exit(EXIT_FAILURE);
        }
    }
    std::ofstream ofs(filename + ".bmp", std::ios::binary);
    if (!ofs){
        std::cerr << "In saveAsBMP function..." << std::endl;
        std::cerr << "Can't open the file." << std::endl;
        exit(EXIT_FAILURE);
    }
    //文件头
    char fileHeader[14] = {0};
    fileHeader[0] = 'B';
    fileHeader[1] = 'M';
    size_t fileSize = 14 + 40 + rows * cols * 3;
    fileHeader[2] = fileSize & 0xff;
    fileHeader[3] = (fileSize >> 8) & 0xff;
    fileHeader[4] = (fileSize >> 16) & 0xff;
    fileHeader[5] = (fileSize >> 24) & 0xff;
    fileHeader[10] = 54;
    ofs.write(fileHeader, 14);
    //信息头
    char infoHeader[40] = {0};
    infoHeader[0] = 40;
    infoHeader[4] = cols & 0xff;
    infoHeader[5] = (cols >> 8) & 0xff;
    infoHeader[6] = (cols >> 16) & 0xff;
```

```

infoHeader[7] = (cols >> 24) & 0xff;
infoHeader[8] = rows & 0xff;
infoHeader[9] = (rows >> 8) & 0xff;
infoHeader[10] = (rows >> 16) & 0xff;
infoHeader[11] = (rows >> 24) & 0xff;
infoHeader[12] = 1;
infoHeader[14] = 24;
ofs.write(infoHeader, 40);
//数据
char* data = new char[rows * cols * 3];
for (size_t i = 0; i < rows; i++){
    for (size_t j = 0; j < cols; j++){
        data[(i * cols + j) * 3] = this->data[(i * cols + j) * 3];
        data[(i * cols + j) * 3 + 1] = this->data[(i * cols + j) * 3 + 1];
        data[(i * cols + j) * 3 + 2] = this->data[(i * cols + j) * 3 + 2];
    }
}
ofs.write(data, rows * cols * 3);
ofs.close();
delete[] data;
}

```

对于loadFromBMP方法，方法会将保存为bmp格式的RGB图像转换为三通道矩阵。它首先读取BMP文件的文件头和信息头，并检查文件头是否正确和信息头是否指示这是一个24位BMP文件。如果不是，它会终止程序并输出错误消息。然后，它使用信息头中的图像大小信息来设置矩阵行数、列数和通道数，分配内存来存储图像数据。接下来，它读取BMP文件中的图像数据，并将其复制到存储图像数据的内存中。最后，它关闭文件并释放临时内存。

```

template <typename T>
void Mat<T>::loadFromBMP(const std::string& filename){
    std::ifstream ifs(filename + ".bmp", std::ios::binary);
    if (!ifs){
        std::cerr << "In loadFromBMP function..." << std::endl;
        std::cerr << "Can't open the file." << std::endl;
        exit(EXIT_FAILURE);
    }
    //文件头
    char fileHeader[14] = {0};
    ifs.read(fileHeader, 14);
    if (fileHeader[0] != 'B' || fileHeader[1] != 'M'){
        std::cerr << "In loadFromBMP function..." << std::endl;
        std::cerr << "The file is not a bmp file." << std::endl;
        exit(EXIT_FAILURE);
    }
    //信息头
    char infoHeader[40] = {0};
    ifs.read(infoHeader, 40);
    if (infoHeader[0] != 40){
        std::cerr << "In loadFromBMP function..." << std::endl;
        std::cerr << "The file is not a bmp file." << std::endl;
        exit(EXIT_FAILURE);
    }
    if (infoHeader[14] != 24){
        std::cerr << "In loadFromBMP function..." << std::endl;
    }
}

```

```

        std::cerr << "The file is not a 24-bit bmp file." << std::endl;
        exit(EXIT_FAILURE);
    }
    rows = *(int*)(infoHeader + 8);
    cols = *(int*)(infoHeader + 4);
    channels = 3;
    size_t size = rows * cols * channels;
    data = new T[size];
    dataStart = data;
    dataEnd = data + size;
    ref_count = new size_t(1);
    //数据
    char* data = new char[rows * cols * 3];
    ifs.read(data, rows * cols * 3);
    for (size_t i = 0; i < rows; i++){
        for (size_t j = 0; j < cols; j++){
            this->data[(i * cols + j) * 3] = data[(i * cols + j) * 3];
            this->data[(i * cols + j) * 3 + 1] = data[(i * cols + j) * 3 + 1];
            this->data[(i * cols + j) * 3 + 2] = data[(i * cols + j) * 3 + 2];
        }
    }
    ifs.close();
    delete[] data;
}

```

对于rgbtogray方法，该方法实现了将彩色图像转换为灰度图像的功能。首先，它检查输入图像是否是彩色图像（即通道数是否为3）。如果不是，它会抛出异常。然后，它创建一个新的灰度图像，并使用以下公式计算灰度值：

$$\text{val} = \text{data}[i * \text{cols} * \text{channels} + j * \text{channels}] * 0.299 + \text{data}[i * \text{cols} * \text{channels} + j * \text{channels} + 1] * 0.587 + \text{data}[i * \text{cols} * \text{channels} + j * \text{channels} + 2] * 0.114$$

这是一种标准的灰度转换方法，其中权重0.299、0.587、0.114分别用于红、绿、蓝通道。最后，它将灰度值分配给新图像的所有通道，并返回新图像。

```

Mat<T> gray(rows, cols, 3);
for (size_t i = 0; i < rows; i++){
    for (size_t j = 0; j < cols; j++){
        T val = (data[i * cols * channels + j * channels] * 0.299 + data[i *
cols * channels + j * channels + 1] * 0.587 + data[i * cols * channels + j *
channels + 2] * 0.114);
        gray.data[i * cols * channels + j * channels] = val;
        gray.data[i * cols * channels + j * channels + 1] = val;
        gray.data[i * cols * channels + j * channels + 2] = val;
    }
}
return gray;

```

5.5 克隆，复制，重塑，转换

在对OPENCV的Mat类进行参考学习后，在本次project中，还针对矩阵的实际使用准备了矩阵函数，分别是克隆，复制，重塑，转换函数


```
Mat<T> clone() const;
Mat<T> copyTo(Mat<T>& dst) const;
Mat<T> reshape(size_t new_cn, size_t new_rows, size_t new_cols) const;
Mat& convertTo(Mat& dst, int rtype, double alpha = 1, double beta = 0) const;
```

clone()实现了矩阵克隆功能。它创建一个新矩阵，并使用输入矩阵的行数、列数和通道数作为参数。然后，它将输入矩阵的每个像素的值复制到新矩阵中。最后，它返回新矩阵。

copyTo方法实现了将矩阵复制到另一个矩阵的功能。它遍历输入矩阵的所有像素，并将每个像素的值复制到目标矩阵的对应像素中。最后，它返回目标矩阵。

reshape实现了图像形状改变的功能。它首先创建一个新矩阵，并使用新的行数、列数和通道数来设置新图像的形状。然后，它使用输入矩阵的数据、数据起始地址和数据结束地址来设置新矩阵的数据。最后，它将新矩阵的引用计数器设置为输入矩阵的引用计数器，并将引用计数器的值加1。这个函数可以用来改变矩阵的形状，但是它并不会分配新的内存来存储数据。相反，它使用输入矩阵的数据来创建新矩阵，并将新矩阵的数据指针指向输入矩阵的数据。

convertTo实现了矩阵数据类型转换的功能。它首先遍历输入矩阵的所有像素，然后使用以下公式计算新的像素值： $\text{dst.data}[i] = (T)(\text{data}[i] * \alpha + \beta)$ 其中，T是目标矩阵的数据类型，alpha是缩放因子，beta是偏移量。最后，它返回目标矩阵。这个函数不会改变矩阵的大小或通道数。它仅仅是将输入矩阵的像素值转换为新的数据类型，并将结果存储在目标矩阵中

5.6 生成基本矩阵

在对OPENCV的Mat类进行参考学习后，在本次project中，还针对矩阵的实际使用准备了一些静态矩阵生成函数，分别能够生成0矩阵，1矩阵和

```
static Mat& zeros(size_t rows, size_t cols, size_t channels);
static Mat& ones(size_t rows, size_t cols, size_t channels);
static Mat& eye(size_t rows, size_t cols, size_t channels);
```

这三个方法实现了创建全零、全一和单位矩阵的功能。首先是 zeros 函数，它创建一个新矩阵，并使用输入的行数、列数和通道数作为参数。然后，它遍历新矩阵的所有像素，并将每个像素设为0。最后，它返回新矩阵。接下来是 ones 函数，它的工作方式与 zeros 函数类似，只是将新矩阵的所有像素设为1。最后是 eye 函数，它创建一个新矩阵，并使用输入的行数、列数和通道数作为参数。然后，它遍历新矩阵的所有像素，并将每个像素设为0。接下来，它遍历新矩阵的所有行，并将每行的对角线上的像素设为1。最后，它返回新矩阵。

Part 3 - Result & Verification

1.创建矩阵

默认构造器实验中分别以整型int以及浮点数float类型创建1通道以及3通道的矩阵并进行元素数组的成员打印以验证正确性。

```
Mat<float> mat3(3, 3, 3);
for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = (i + 1.0) * (j + 1.0);
        mat3(i, j, 1) = (i + 1.0) * (j + 1.0) + 1.0;
        mat3(i, j, 2) = (i + 1.0) * (j + 1.0) + 2.0;
    }
}
cout << mat1 << endl;
```

```
{1 2 3}, {2 3 4}, {3 4 5}
{2 3 4}, {4 5 6}, {6 7 8}
{3 4 5}, {6 7 8}, {9 10 11}
```

```
Mat<int> mat1(3, 3, 1);
for(size_t i = 0; i < mat1.getRows(); i++) {
    for(size_t j = 0; j < mat1.getCols(); j++) {
        mat1(i, j, 0) = i + j;
    }
}
cout << mat1 << endl;
```

```
{0}, {1}, {2}
{1}, {2}, {3}
{2}, {3}, {4}
```

2.复制构造函数以及赋值运算符重载的正确性验证

在该实验中，在创建对象成功的时候显示当前存在的矩阵对象个数；由于复制构造函数与赋值运算符重载需要避免硬拷贝，所以涉及到两个矩阵对于同一块内存空间的共享，所以在两种函数生成时也输出当前矩阵的内存区域被多少个矩阵共享；在析构函数当中，由于在释放对象的时候需要判断该对象的元素数组所占的空间同时被多少个矩阵对象所共用，所以在析构函数当中也输出了相关信息。最终只需要判断输出语句是否符合空间释放的逻辑即可。

由如下结果所示，首先通过默认构造器创建了一个对象mat1，然后对mat1赋值，此时矩阵对象数是1，该区域引用数是1，然后通过复制构造器传入mat1创建了一个对象mat2，此时矩阵对象数是2，该区域引用数是2，之后通过赋值运算符传入mat1创建了一个对象mat3，此时矩阵对象数是3，该区域引用数是3，过后通过默认构造器创建了一个对象mat4，此时矩阵对象数是4，mat4区域引用数是1，最后程序结束时调用析构函数，由于程序运行的次序类似于数据结构中的栈，根据栈先进先出(FIFO)的特性，得知第一个销毁的矩阵对象是最后创建的mat4，此时析构函数释放了mat4所在的内存，然后又销毁了mat3和mat2，最后在销毁mat1时释放了mat1所在的内存。至此，所有的对象以及其对象数组所占的内存空间都被完全释放，说明对矩阵的内存管理是可圈可点的。

```
cout << mat1 << endl;
cout << "mat1.getRefCount() = " << mat1.getRefCount() << endl;
cout << "mat1.getNumMatrices() = " << mat1.getNumMatrices() << endl;
Mat<int> mat2(3, 3, 1);
mat2 = mat1;
cout << mat2 << endl;
cout << "mat2.getRefCount() = " << mat2.getRefCount() << endl;
cout << "mat2.getNumMatrices() = " << mat2.getNumMatrices() << endl;
Mat<int> mat3(mat1);
cout << mat3 << endl;
```

```

cout <<"mat3.getRefCount() = " << mat3.getRefCount() << endl;
cout <<"mat3.getNumMatrices() = " << mat3.getNumMatrices() << endl;
Mat<int> mat4(3, 3, 1);
cout << mat4 << endl;
cout <<"mat4.getRefCount() = " << mat4.getRefCount() << endl;
cout <<"mat4.getNumMatrices() = " << mat4.getNumMatrices() << endl;

```

```

{0}, {1}, {2}
{1}, {2}, {3}
{2}, {3}, {4}
mat1.getRefCount() = 1
mat1.getNumMatrices() = 1
{0}, {1}, {2}
{1}, {2}, {3}
{2}, {3}, {4}
mat2.getRefCount() = 2
mat2.getNumMatrices() = 2
{0}, {1}, {2}
{1}, {2}, {3}
{2}, {3}, {4}
mat3.getRefCount() = 3
mat3.getNumMatrices() = 3
{0}, {0}, {0}
{0}, {0}, {0}
{0}, {0}, {0}
mat4.getRefCount() = 1
mat4.getNumMatrices() = 4
The matrix is free
Destructor called
Destructor called
Destructor called
The matrix is free
Destructor called

```

3.矩阵访问运算符重载的正确性实验

由于本矩阵类已经通过使用重载的访问运算符实现了<<运算符的重载，故通过如下代码即可验证访问运算符重载的正确性，既可以修改矩阵元素，又可以访问矩阵元素。

```

Mat<float> mat3(3, 3, 3);

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = (i + 1.0) * (j + 1.0);
        mat3(i, j, 1) = (i + 1.0) * (j + 1.0) + 1.0;
        mat3(i, j, 2) = (i + 1.0) * (j + 1.0) + 2.0;
    }
}
// 打印矩阵
cout << mat3 << endl;

```

```
{1 2 3}, {2 3 4}, {3 4 5}
{2 3 4}, {4 5 6}, {6 7 8}
{3 4 5}, {6 7 8}, {9 10 11}
```

4.矩阵基本运算正确性检验

在该实验中，我们将检验矩阵加减法的正确性。

```
Mat<float> mat3(3, 3, 3);
Mat<float> mat4(3, 3, 3);

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = 1.0;
        mat3(i, j, 1) = 1.0;
        mat3(i, j, 2) = 1.0;
    }
}

for (size_t i = 0; i < mat4.getRows(); i++) {
    for (size_t j = 0; j < mat4.getCols(); j++) {
        mat4(i, j, 0) = 2.0;
        mat4(i, j, 1) = 2.0;
        mat4(i, j, 2) = 2.0;
    }
}

// 打印矩阵
cout << mat3 << endl;
cout << mat4 << endl;
// 矩阵相加
cout << mat3 + mat4 << endl;
// 矩阵相减
cout << mat3 - mat4 << endl;
```

```
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{3 3 3}, {3 3 3}, {3 3 3}
{3 3 3}, {3 3 3}, {3 3 3}
{3 3 3}, {3 3 3}, {3 3 3}
{-1 -1 -1}, {-1 -1 -1}, {-1 -1 -1}
{-1 -1 -1}, {-1 -1 -1}, {-1 -1 -1}
{-1 -1 -1}, {-1 -1 -1}, {-1 -1 -1}
```

5.矩阵乘法正确性检验

在该实验中，我们将检验矩阵乘法的正确性。

```
// 矩阵相乘
cout << mat3 * mat4 << endl;
Mat <float> mat5(3, 1, 3);
Mat <float> mat6(1, 3, 3);
```

```

for (size_t i = 0; i < mat5.getRows(); i++) {
    for (size_t j = 0; j < mat5.getCols(); j++) {
        mat5(i, j, 0) = 1.0;
        mat5(i, j, 1) = 1.0;
        mat5(i, j, 2) = 1.0;
    }
}
for (size_t i = 0; i < mat6.getRows(); i++) {
    for (size_t j = 0; j < mat6.getCols(); j++) {
        mat6(i, j, 0) = 2.0;
        mat6(i, j, 1) = 2.0;
        mat6(i, j, 2) = 2.0;
    }
}
cout << mat5 << endl;
cout << mat6 << endl;
cout << mat5 * mat6 << endl;

```

```

{6 6 6}, {6 6 6}, {6 6 6}
{6 6 6}, {6 6 6}, {6 6 6}
{6 6 6}, {6 6 6}, {6 6 6}
{1 1 1}
{1 1 1}
{1 1 1}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}

```

6.ROI 相关操作正确性检验

在该实验中，我们将检验ROI 相关操作的正确性。

```

Mat<float> mat3(3, 3, 3);
Mat<float> mat4(3, 3, 3);

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = 1.0;
        mat3(i, j, 1) = 1.0;
        mat3(i, j, 2) = 1.0;
    }
}
mat4 = mat3.ROI(1, 1, 1, 1);
cout << mat3 << endl;
cout << mat4 << endl;

```

```

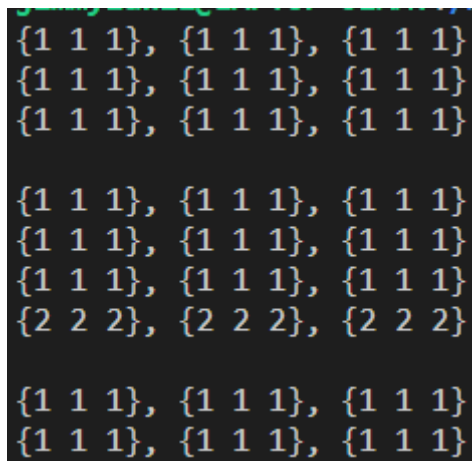
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}

```

7.在矩阵底部添加或删除元素的正确性检验

```
Mat<float> mat3(3, 3, 3);
for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = 1.0;
        mat3(i, j, 1) = 1.0;
        mat3(i, j, 2) = 1.0;
    }
}
float *data = new float[9];
for (size_t i = 0; i < 9; i++) {
    data[i] = 2.0;
}
cout << mat3 << endl;
cout << endl;

mat3.append(data, 1, 3, 3);
cout << mat3 << endl;
cout << endl;
mat3.subtract(2);
cout << mat3 << endl;
```



The image displays a 3x3 grid of 3x3 matrices. The first two rows show matrices with all 1s. The third row shows matrices with 1s in the first two rows and 2s in the third row, representing the state after the append and subtract operations.

8.矩阵合并正确性验证

```
Mat<float> mat3(3, 3, 3);
Mat<float> mat4(3, 3, 3);
Mat<float> mat5(3, 3, 3);
Mat<float> mat6(3, 3, 3);

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = 1.0;
        mat3(i, j, 1) = 1.0;
        mat3(i, j, 2) = 1.0;
    }
}
for (size_t i = 0; i < mat4.getRows(); i++) {
    for (size_t j = 0; j < mat4.getCols(); j++) {
        mat4(i, j, 0) = 2.0;
        mat4(i, j, 1) = 2.0;
```

```

        mat4(i, j, 2) = 2.0;
    }
}

mat5 = mat3.merge_vertical(mat4);
cout << mat5 << endl;
mat6 = mat3.merge_horizontal(mat4);
cout << mat6 << endl;

```

```

{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{2 2 2}, {2 2 2}, {2 2 2}
{1 1 1}, {1 1 1}, {1 1 1}, {2 2 2}, {2 2 2}, {2 2 2}
{1 1 1}, {1 1 1}, {1 1 1}, {2 2 2}, {2 2 2}, {2 2 2}
{1 1 1}, {1 1 1}, {1 1 1}, {2 2 2}, {2 2 2}, {2 2 2}

```

9.矩阵通道的分离/合并正确性验证

```

Mat<float> mat3(3, 3, 3);

for (size_t i = 0; i < mat3.getRows(); i++) {
    for (size_t j = 0; j < mat3.getCols(); j++) {
        mat3(i, j, 0) = 1.0;
        mat3(i, j, 1) = 1.0;
        mat3(i, j, 2) = 1.0;
    }
}

cout << mat3 << endl;
vector<Mat<float>> mv;
mat3.split(mv);
cout << mv[0] << endl << endl;
cout << mv[1] << endl << endl;
cout << mv[2] << endl << endl;
mat3.merge(mv);
cout << mat3 << endl;

```



```
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1}, {1}, {1}
{1}, {1}, {1}
{1}, {1}, {1}

{1}, {1}, {1}
{1}, {1}, {1}
{1}, {1}, {1}

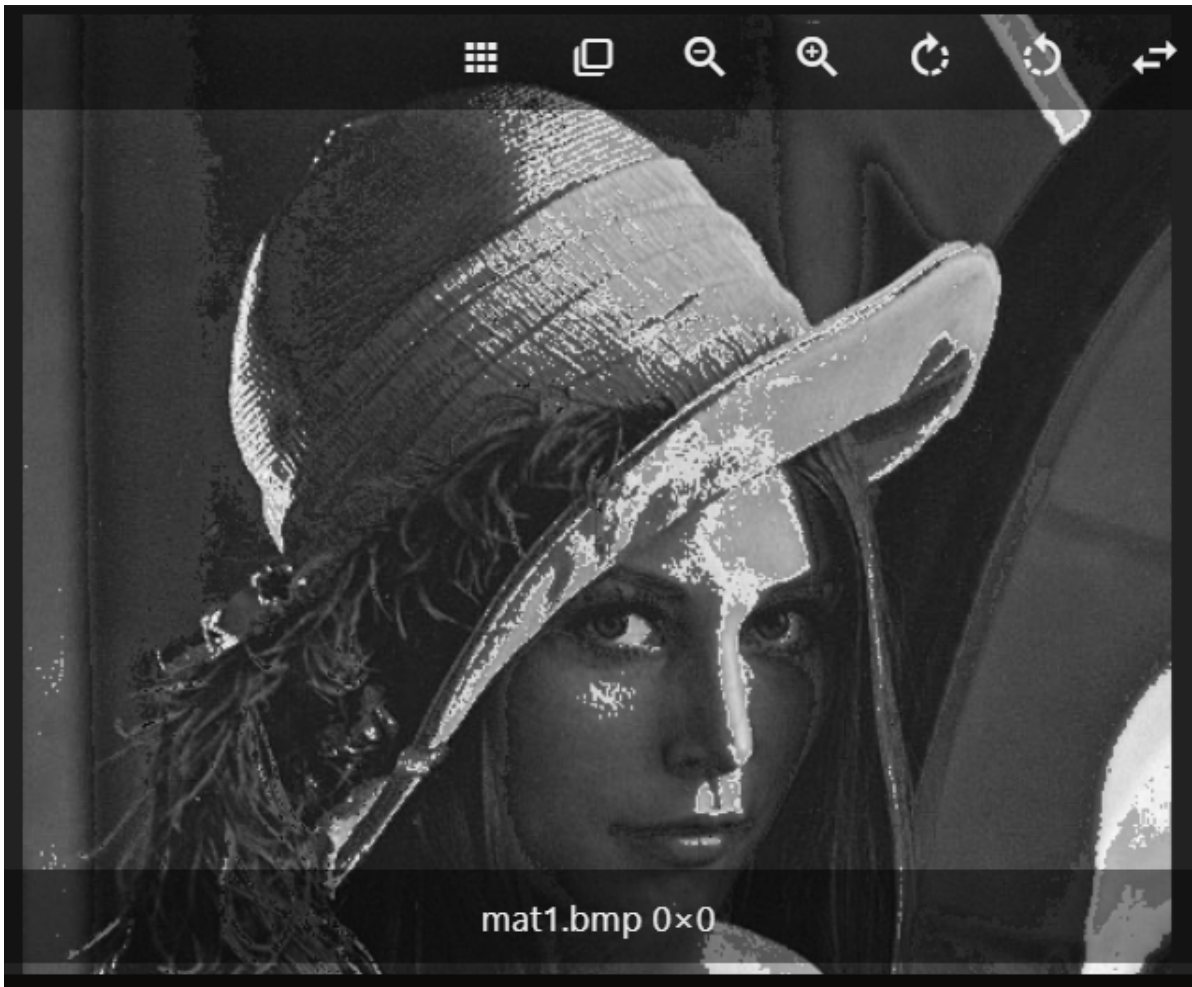
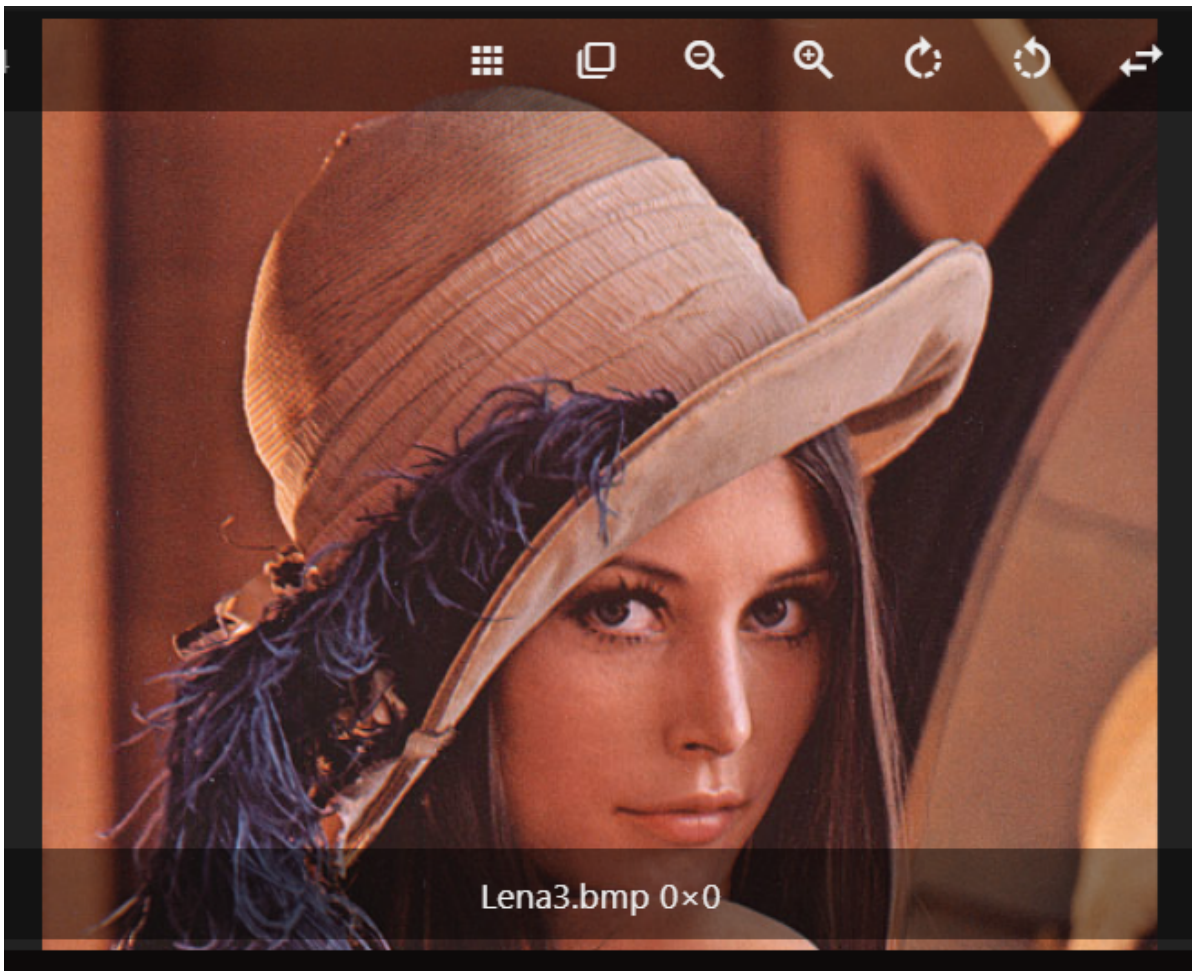
{1}, {1}, {1}
{1}, {1}, {1}
{1}, {1}, {1}

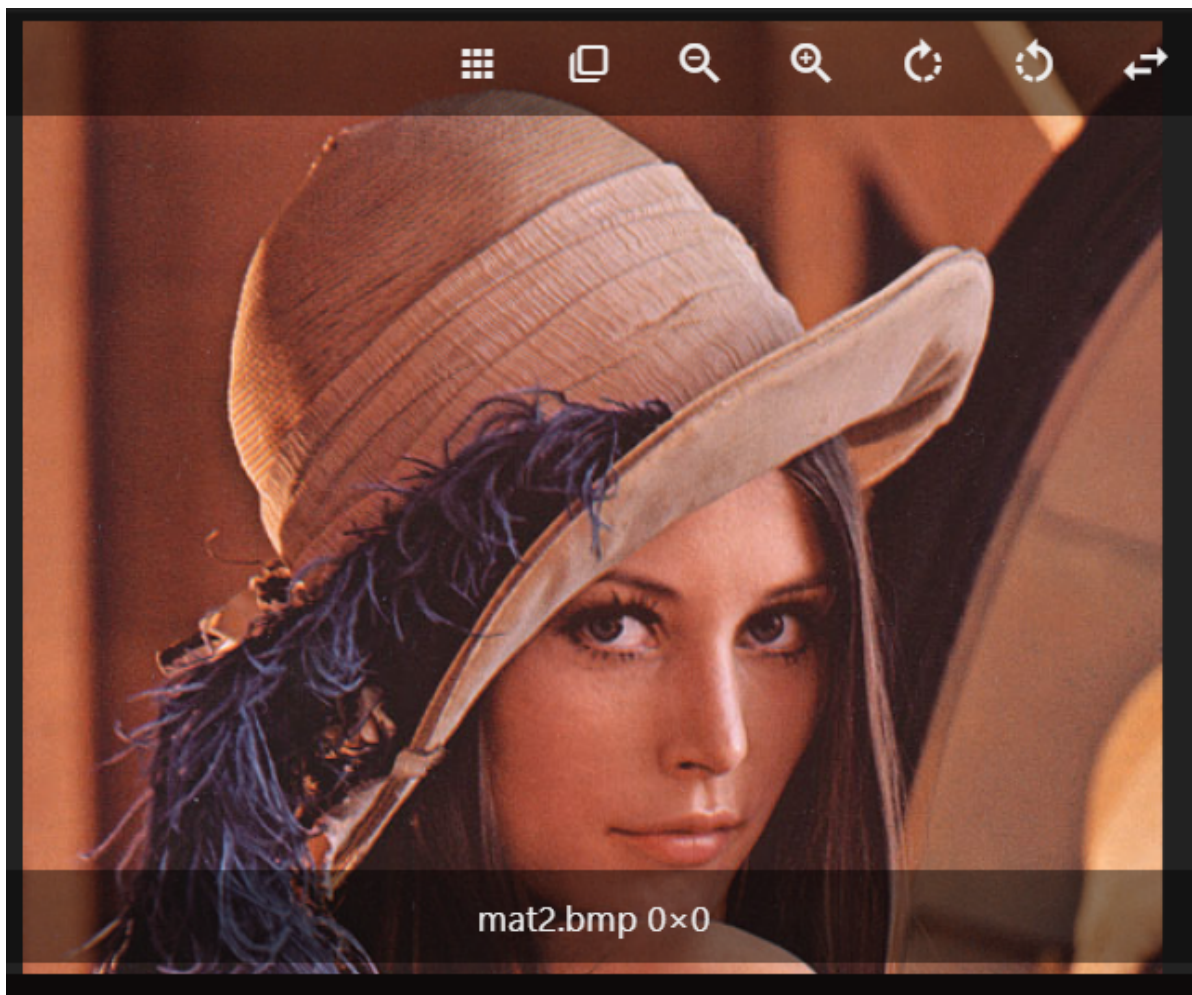
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
{1 1 1}, {1 1 1}, {1 1 1}
```

10.IO函数正确性验证

本实验主要验证图像函数的正确性。

```
Mat<float> mat1(512, 512, 3);
Mat<float> mat2(256, 256, 3);
mat1.loadFromBMP("Lena3");
mat1.cvtColor().saveAsBMP("mat1");
mat2 = mat1;
mat2.saveAsBMP("mat2");
```





11.Hard Copy 与 Soft Copy 时间差异比较

随着矩阵规模的增大，需要创建子矩阵的时间也会急速增多，甚至在矩阵规模为 4096×4096 的时候，硬拷贝的时间是软拷贝时间的数百倍，这是一个惊人的量级。在实际运用矩阵进行图像处理的时候，面对的常常是规模极大的矩阵，通过本次实验也能更加直观地感受到软拷贝在实际应用当中发挥着重要的作用。

```
size_t size = 4096;
Mat<float> mat1(size, size, 3);
Mat<float> mat2(size, size, 3);

struct timeval start, end;
gettimeofday(&start, NULL);
mat2 = mat1;
gettimeofday(&end, NULL);
printf("soft copy Time cost: %ld us\n",
       (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec));

gettimeofday(&start, NULL);
mat2 = mat1.clone();
gettimeofday(&end, NULL);
printf("hard copy Time cost: %ld us\n",
       (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec -
start.tv_usec));
```

| 软拷贝时间/us | 矩阵规模 | 硬拷贝时间/us |
|----------|-----------|----------|
| 1 | 32*32 | 4 |
| 29 | 256*256 | 204 |
| 58 | 512*512 | 1019 |
| 258 | 1024*1024 | 4319 |
| 143 | 2048*2048 | 16843 |
| 766 | 4096*4096 | 57010 |

Part 4 - Difficulties&Solutions

1.ROI设计

一开始做project的时候，对于设计ROI操作没有思路，后来在仔细查阅openCV的源码设计和issue帖子，并且请教于老师后，明白了cv::Mat类中data，dataStart，dataEnd，dataLimit之间的关系，于是借鉴opencv的做法融入到本次project的矩阵类中

2.数据数组未初始化

在做实验验证矩阵运算时，一开始遇到了十分离奇的bug，矩阵乘法一直无法得到正确结果，甚至在修改不相关代码后程序结果会变化，后来经过刻骨铭心的一段debug过程后，发现是因为少参数的默认构造器中新了新数据数组，但是没有给新数据数组置零，导致结果矩阵的初始值为随机数，在修改默认构造器后成功解决了这个问题。

Part 5 - Conclusion

这次project5是project3的c++版本，也是升级版，因为在各个维度上这次project都有着更高的要求，比如软拷贝，软拷贝带来的内存管理难题，更多更复杂的矩阵函数，ROI矩阵的实现，还有运算符重载和模板类的使用，综合地考察了我们前面几乎所有课程的知识。

在本次project中，我对于软拷贝和引用传递有了更深的理解，也初步具有处理其带来的内存管理问题的能力，在查阅资料的过程中，对OPENCV的Mat类和ROI矩阵的实现有了更深的理解，在码了1500行后加深了对模板和运算符重载的理解。

当然也很多没有实现的功能，比如更系统全面的错误捕捉和警示，使用smartptr，实现内存分配器，更多的图像处理函数等等，希望在日后能你补这些空缺。

谢谢老师的阅读！