CS205 C/ C++ Programming - Project4

name: 廖铭骞

SID: 12012919

Part1 Analysis

Project 基本要求

1. 设计一个矩阵类，其中包含矩阵的基本数据信息（行数、列数、channel数...）
2. 矩阵类支持不同的数据类型，即元素的数据类型应该多元化
3. 尽可能避免大块内存的复制，使用引用传值，以及完成很好的内存管理，避免内存溢出以及动态分配与释放内存
4. 重载运算符完成矩阵相关操作（赋值运算符，判断相等运算符，矩阵加法运算符，矩阵减法运算符，以及矩阵乘法运算符等等）
5. 实现ROI （OpenCV中感兴趣区域）
6. 在X86架构以及ARM服务器中运行程序，并且比较差异

模板类的编写与使用

由于矩阵里面存储的类型多样，而不仅仅限于float或者int类型，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。一种实现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，其次，在每个程序中只能使用这种技术生成一种矩阵类，即不能让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。

所以本程序在此处使用C++的模板类实现这一功能，首先在基础的程序当中，我先使用了float类型作为基础类型实现了矩阵类的基本方法，最后再进行模板类的替换，将原本基于float类型的矩阵类进一步抽象，抽象成为模板类。进一步地，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原有的函数。

根据课上所学知识——模板并不是具体的类和成员函数，它们类似于指导C++编译器的蓝图，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实现文件当中。在此处将所有模板信息都整合放在头文件当中。

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

```
Matrix<float> mat;
```

则会使得编译器使用float类型替换模板中所有的类型参数，在实际使用发现，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。

矩阵类的分析与设计

一个矩阵当中首先应该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，并且还应该有存储这个矩阵对应元素的元素数组nums，由于对于一个矩阵而言一个元素可能有不同的通道，由于对于同一个元素而言，如果要进行不同通道值的访问，如果需要跳过一段又一段大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合理性，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。对于同一块数组区域，可以有多个矩阵共用这一块区域，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。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。除此之外，一个矩阵还应该维护一个ROI区域的起始点ROIBeginRow,ROIBeginCol以及ROI的行数ROIRow以及列数ROICol`。由于在矩阵的实际运算当中可能涉及到不同数据类型的运算，所以在本程序当中使用模板类对该问题进行解决，该矩阵类所包含的类方法在下一小结当中进行阐述。

模板矩阵类函数的分析与编写

默认构造器

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，确保行列数为正数，通道数最多为三，最小为一，以及ROI区域被母矩阵真包含，不会越界。最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。在最后给表示当前存在矩阵个数的静态变量num_matrices 值加一。

注意此处对于元素数组的初始化，由于本次矩阵运算将实现多通道矩阵的运算，所以矩阵的每一个元素都可能有至多三个值，代表三个通道，所以此处为元素数组分配的内存空间size的计算公式如下：

$$size = row * col * channel$$
$$(channel：通道数 \quad row：矩阵的行数 \quad col：矩阵的列数)$$

函数头如下：

```
template<typename T>
Matrix<T>::Matrix(int row, int col, int channel, T *nums, int
ROIBeginRow, int ROIBeginCol, int ROIRow, int ROICol);
```

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();
```

但在后续的实验中发现，由于对于元素数组传入的变量是一个数组指针，但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将传入数组的值复制一遍作为自己的元素数组，这样使得传入数组时默认函数更加地安全可靠，避免隐式地影响多个矩阵元素数组的值。

由于传入大规模的矩阵经常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以及文件包含数组的长度获取其中元素信息来创建矩阵的方法。

通过传入文件创建矩阵的构造器声明如下(fin 是传入的文件流，size 描述了文件中包含元素的个数)：

```
template<typename T>
Matrix<T>::Matrix(int row, int col, int channel, std::ifstream &
fin, int size, int ROIBeginRow, int ROIBeginCol, int ROIRow, int
ROICol, int *ref_count, int span){
```

大体上的方法与默认构造器一致，都需要根据传入的参数对矩阵对象进行初始化。

对于传入的参数而言，除了进行常规的检查之外，还需要额外判断传入的文件包含的数组元素个数是否与矩阵的大小相一致

```
    //对于传入的文件当中包含的数组大小进行检查
    if(row * col * channel != size * size){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The input array's size should equal to the
size of matrix's element array." << std::endl;
        exit(EXIT_FAILURE);
    }
```

如果一致，则下一步进行矩阵元素数组对应元素的赋值。

```
    this -> nums = new T[size];
    int pos = 0;
    while(!fin.eof()){
        std::string line;
        getline(fin, line);
        std::stringstream ss(line);
        T f;
        while(ss >> f){
            this -> nums[pos++] = f;
        }
    }
```

经过检验，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。

复制构造函数

在之前的实验当中总结发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浅复制，即仅仅复制成员的值，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。如在按值传递给某一函数的时候，将会把该对象按值传递给一个临时变量，在该函数调用完毕之后，临时变量被释放，这将导致该地址的内存也会被释放，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。所以，定义复制构造函数十分必要。

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，形参就是实参的一个别名，避免了大段内存的复制，使用的是原始数据而非副本。

同样地，需要对传入的变量进行检测，如果是空值，则提示错误后退出程序

```
if(mat == NULL){
        std::cerr << "In copy constructor" << std::endl;
        std::cerr << "The input matrix should be valid." <<
std::endl;
    }
```

之后便一一对变量进行初始化，对于矩阵的元素数组，则需要一一对里面的值进行深复制

```
//对复制源矩阵的元素数组进行深复制
    int ele_num = mat.row * mat.col * mat.channel;
    this -> nums = new T[ele_num];//动态分配内存
    for(int i = 0; i < ele_num; i++){
        this -> nums[i] = mat.nums[i];
    }
```

但是这样进行深复制的效率非常低，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，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的同时，将表示引用次数的指针代表的值加一，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被释放两次的情况。改进的代码如下：

```
    this -> ref_count = mat.ref_count;
    this -> ref_count[0] ++;//将引用该内存块的次数加一
    this -> nums = mat.nums;//将新矩阵的元素数组的指针指向原矩阵元素数
组的指针
```

最后将表示当前存在矩阵类对象个数的变量值加一。

```
num_matrices ++;
```

析构函数

对于矩阵类的析构函数，最主要的工作便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判断表示当前内存区域引用次数的值是否为1，如果为1，则说明该区域只有一个矩阵对象在使用，此时应该直接释放该区域，否则将该值减一，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减一。

```
//析构函数
template<typename T>
Matrix<T>::~Matrix(){
    num_matrices--;
    if(this -> ref_count[0] == 1){
        delete[] this -> ref_count;
        this -> ref_count = nullptr;
        delete[]nums;
        this -> nums = nullptr;
    }

    else
        this -> ref_count[0]--;
}
```

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，并且只执行对赋值源对象的非静态成员变量的浅复制，对于成员指针变量，与默认的复制构造函数相同道理，会只将指针代表的地址复制给待复制对象，这样也会出现和未重载复制构造函数同样的问题，即可能会因为同一块内存空间被释放了两次而使程序崩溃，为了避免这一问题，使得程序更加稳定安全地运行，此处重载赋值运算符。

首先需要判断传入的引用对象是否为空，即进行程序安全性的检查。确认传入对象有效之后，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，结束程序。如果不相同，则先判断待赋值的矩阵原先的元素数组所在的内存区域被多少个矩阵对象共用，如果只有自身共用，则释放原有的数组空间，如果不只是自己使用这块内存空间，则将使用该内存空间的对象值减一，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，当矩阵元素非常多的时候，这将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，则完成了赋值运算符的重载。

访问元素运算符()的重载

为了更便于矩阵元素以及其通道元素的访问，本程序额外重载了()运算符，通过括号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，由于在本程序当中，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 nums[r * span + c * channel + ch - 1];
```

其中`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，而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，来到指定的目标列，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更改。

矩阵运算符重载

由于矩阵的加法、减法以及除法运算机制相同，所以此处以矩阵的加法为例介绍在本项目中矩阵加减法以及除法运算符的重载。

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。

其次，+运算符要求返回一个全新的矩阵，其中的元素为两源矩阵元素逐个相加的结果，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。

```
    mat.nums = new T[row * col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
this -> nums[i * this -> span + j *

              this -> channel + ch -1] + num;
                // mat(i, j, ch) = (*this)(i, j, ch) + num; 由于访问
运算符重载只允许非const调用，以及返回非const， 不符合矩阵加法决不允许在
函数当中不小心改变原矩阵值的特性，故采用直接运算访问的方式访问。
            }
        }
    }
return mat;
```

为了防止传入空指针对程序运行稳定性造成破坏，在正式开始运算之前，同样地需要进行实参为空情况的判断，如果实参不为空，并且两个矩阵的大小互相匹配，则在函数内部创建一个新的矩阵对象。

经过逐个的加法运算，最终返回新的矩阵对象。

对于+= 运算符的重载，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，最终返回this指针指向的对象即可。但是需要注意，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，这并不符合规则，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，否则需要重新开辟一个新的空间，并将新的值赋值给新的元素数组，并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，然后将代表引用自身数组空间的值设置为1。

核心代码如下：

```
    int ele_num = row * col * channel;//总的元素个数
    if(this -> ref_count[0] == 1){//如果引用次数为1，则直接改变原数组的元素值

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 0; ch < channel; ch++){
                    *this(i, j, ch) = *this(i, j, ch) + mat(i, j, ch);
                }
            }
        }
    }else{//如果引用数不为1，则需要重新开辟一块新的空间用于存储新的元素值
        this -> ref_count[0] --;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];//开辟一块新的空间存储元素数组
        int pos = 0;

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 0; ch < channel; ch++){
                    n[pos++] = *this(i, j, ch) + mat(i, j, ch);
                }
            }
        }
        this -> nums = n;
    }
    return *this;
```

进一步地，对于代表单个整数或浮点数的变量而言，存在自增运算符可以使得该变量的值+1或者-1，而对于矩阵的元素而言，也可以将每一个元素通过自增运算符加一或者减一，这一操作可以通过重载自增运算符解决。

类似于课上所学对于类自增运算符的重载，对于矩阵对象的自增运算符也可以通过依次将每一个矩阵元素的值自增实现。

同样地，需要判断该矩阵的内存区域被多少个矩阵对象共用，如果只有自己使用，则可以直接改变该区域的值，否则也需要另外开辟一个新的空间去存储新的自增之后的值。核心代码如下所示：

```
    for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 0; ch < channel; ch++){
                    *this(i, j, ch) = *this(i, j, ch) + 1;
                }
            }
        }
```

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。

友元函数

友元函数的声明虽然位于类内部，可以访问并操作类内部的成员变量，即与成员函数的访问权限相同，但是其并不属于类的成员函数。

那么友元函数是否违法面向对象编程的原则——封装呢？通过查阅资料，一种解释是友元函数并不悖于面向对象的原则，因为决定哪一个函数是友元函数是由类的声明决定的，因此类声明依然控制了函数的访问权限，这样就可以将友元函数看作是表达类接口的另一种不同机制。

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。

```
template<typename T>
Matrix<T> operator+(T num, const Matrix<T>& mat){
    return mat + num;//使用运算符重载即可完成
}
```

此时如果想要遍历查看矩阵的每一个元素的值，每一个值的通道值都通过花括号括起来，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，在重复调用，这样很不直观，而友元函数的另一个常用点就是重载 << 运算符，使之能够与cout一起显示矩阵的每一个元素以及通道的值。

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。由于已经重载了矩阵访问运算符，矩阵元素的访问变得更加方便也更加直观，具体实现代码如下：

```
//重载<<运算符
template<class T>
std::ostream & operator<<(std::ostream &os, Matrix<T>& mat){
    int ch = mat.channel;
    int r = mat.row;
    int c = mat.col;
    os << "[";
    for(int i = 0; i < r; i++){
        for(int j = 0; j < c; j++){
            os << "{";
            for(int k = 1; k <= ch; k++){
                os << mat(i, j, k);
                if(k != ch)  os << " ";
            }
            os << "}";
            if(j != c - 1) os << ", ";
        }
        if(i != r - 1) os << "\n";
    }
    os << "]";
```

```
    return os;
}
```

对于一个通道数为2的矩阵mat2使用重载之后的<<运算符对其进行元素数组的输出，实验代码如下所示：

```
int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = i + 1;
    }

    Matrix<int> mat2(3, 5, 2, nums1);//行数为3， 列数为5， 通道数为2，以数组nums1作为元素数组
    cout << mat2 << endl;
```

打印结果如图所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
[{1 2}, {3 4}, {5 6}, {7 8}, {9 10}
{11 12}, {13 14}, {15 16}, {17 18}, {19 20}
{21 22}, {23 24}, {25 26}, {27 28}, {29 30}]
```

通过花括号将同一通道的值括起来，这样更便于区分。

矩阵的转置运算符重载

通过参考OpenCV的实现方法，发现在实际应用当中对于矩阵的转秩运算也十分重要，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。

通过使用上面实现的矩阵访问运算符，可以忽略很多复杂的计算来更加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秩赋值，核心代码如下：

```
for(int i = 0; i < mat.row; i++){
        for(int j = 0; j < mat.col; j++){
            for(int k = 1; k <= mat.channel; k++){
                mat(i, j, k) = this -> nums[j * this -> span + i *
this -> channel + k - 1];
            }
        }
    }
```

最后返回新创建的矩阵mat

矩阵乘法的初步实现

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相乘，矩阵和数字相乘的方法和加减法运算符重载类似，只需要将矩阵的元素数组当中每一个元素与该数字相乘即可。

在实现两个矩阵相乘的时候，首先需要做的依然是判断输入是否合法，如判断传入矩阵是否为空值，判断两个矩阵的大小以及通道数是否相同，确保参数没问题了才可以进行矩阵的乘法。

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乘法。

核心代码如下：

```
    for(int ch = 0; ch < channel; ch++){//不同通道数的切换
        for(int k = 0; k < col; k++){
            for(int i = 0; i < row; i++){
                for(int j = 0; j < col; j++){
                    res(i, j, ch) += m(i, k, ch) * src(k, j, ch);
                }
            }
        }
    }
```

在矩阵底部添加若干行数据

在参考OpenCV方法的时候发现有在矩阵底部添加若干行数据的函数，于是思考对该方法的实现

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。首先进行的依然是对传入的参数进行检查，能在原矩阵底部添加数组当且仅当满足该数组的列数以及通道数之和与原矩阵相同的条件。其次，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一一赋值到新数组当中，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，如果只有自己使用则释放原有的空间，否则将原有的引用值减一，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。

核心代码如下所示：

```
    int src_ele_num = row * col * channel;//原矩阵元素
    int tar_ele_num = r * c * ch;//新传入元素数组的大小数组的大小
    T * arr = new T[src_ele_num + tar_ele_num];//创建新元素数组，大小为两个数组大小之和
    int pos = 0;
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 0; ch < channel; ch++){
                arr[pos++] = *this(i, j, ch);
            }
        }
    }
    for(int j = 0; j < tar_ele_num; j++){
        arr[i++] = nums[j];//在新数组后方赋值为新传入的数组
    }
    if(this -> ref_count[0] == 1){
        delete[] this -> nums;//防止内存泄漏，释放源矩阵原有的元素数组所占空间
    }else{//将原有引用值加一，并将新的引用值设置为1
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count = 1;
    }

    this -> nums = arr;//将矩阵对象的元素数组指向新开辟的内存区域
    this -> row = this -> row + r;
    return *this;
```

在矩阵底部删除若干行数据

通过阅读OpenCV实现的方法，发现除了在矩阵底部添加若干行数据之外，有时候也需要进行在矩阵的底部删除若干行数据的操作，在实际的图像应用方面，该方法可以应用于删除一幅图像当中不必要的冗余信息。经过思考，只需要传入希望删除数据的行数即可，首先需要检查传入的行数是否大于原矩阵对象的行数，以及传入的行数是否大于零，即检查数据是否合法。如果都合法，则与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。

核心代码如下所示：

```
    int ele_num = (row - r) * col * channel;//新数组大小
    T* arr = new T[ele_num];
    int pos = 0;
    for(int i = 0; i < row - r; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 0; ch < channel; ch++){
                arr[pos++] = *this(i, j, ch);
            }
        }
    }
    if(this -> ref_count[0] == 1){
        delete[] this -> nums;//防止内存泄漏，释放源矩阵原有的元素数组所占空间
    }else{//将原有引用值加一，并将新的引用值设置为1
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count = 1;
    }
    this -> nums = arr;//赋值
    this -> row -= r;
    return *this;
```

矩阵的合并（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。下面实现矩阵的纵向合并。

和上文中提到的在矩阵底部添加若干行数据类似，在进行矩阵合并的时候首先关注的应该是传入的矩阵能否与主调矩阵对象进行合并，对于纵向合并，应该确认的是传入矩阵的列数以及通道数是否与源矩阵一致，如果一致，才能进行合并。当然也要检查传入矩阵是否为空以及其元素数组是否为空。

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ROI区域则设置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即可。

核心代码如下：

```
    Matrix<T> res;
    res.row = row1 + row2;
    res.col = col;
    res.channel = channel;
    res.nums = new T[row * col * channel];

    int pos = 0;
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 0; ch < channel; ch++){
                res.nums[pos++] = *this(i, j, ch);
            }
        }
    }

    for(int i = 0; i < mat.row; i++){
        for(int j = 0; j < mat.col; j++){
            for(int ch = 0; ch < mat.channel; ch++){
                res.nums[pos++] = mat(i, j, ch);
            }
        }
    }
```

矩阵的合并（横向）

与矩阵的纵向合并相比，横向合并对于传入参数的检查有所不一样，横向合并能够进行当且仅当两个矩阵的行数要相等。所以除了检查传入矩阵以及其元素数组是否为空值之外，还要判断两个矩阵的行数是否一致。

横向合并两个矩阵主要的难点在于不能顺序对其进行赋值操作，需要逐行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。一开始的实现如下所示：

```
    Matrix<T> res;
    res.row = row;
    res.col = col + mat.col;
    res.channel = channel;
    res.span = res.col * res.channel;
    res.nums = new T[row * res.col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col ; j++){
            for(int ch = 0; ch < channel; ch++){
                res(i, j, ch) = *this(i, j, k);
            }

        }

        for(int j = 0; j < mat.col ; j++){
            for(int ch = 0; ch < channel; ch++){
                res(i, col + j, ch) = mat(i, j, ch);
            }
        }
    }

    return res;
```

这样的实现虽然能够实现矩阵的横向合并，但是在每一层的循环当中都要重复计算两个矩阵每一行的元素个数，这样实现的效率较低，所以，进一步地可以将每一行的元素个数在循环外计算出来，从而在每一层循环当中减少对重复元素的计算量，这样可以提高一部分效率，改变之后的代码如下所示：

```
Matrix<T> res;
```

```
    res.row = row;
    res.col = col + mat.col;
    res.channel = channel;
    res.nums = new T[row * res.col * channel];
    int size = res.col * channel;
    int size1 = col * channel;
    int size2 = mat.col * channel;
    for(int i = 0; i < row; i++){
        int pos = 0;
        int row_number1 = i * size;
        int row_number2 = i * size1;
        int row_number3 = i * size2;
        for(int j = 0; j < col * channel; j++){
            res.nums[row_number1 + pos] = nums[row_number2 + j];
            pos++;
        }

        for(int j = 0; j < mat.col * channel; j++){
            res.nums[row_number1 + pos] = mat.nums[row_number3 +
j];
            pos++;
        }
    }

    return res;
```

通过事先计算好对应的行数，可以减少在循环当中重复计算该值，从而提高计算的效率。

向量点乘

由于计算两个向量点乘结果并不依赖于一个具体的对象，所以此处设置为静态类函数。

输入两个代表向量的数组，经过参数检查确定两个向量均为有效的输入（行数不小于零，两向量大小相同，传入的指针不为空）之后进行向量点乘运算，运算公式如下：

$$A = [a_1, a_2, a_3, \ldots, a_n]$$
$$B = [b_1, b_2, b_3, \ldots, b_n]$$
$$A \cdot B = \sum_{i=1}^{n} a_i * b_i$$

最终返回运算得到的结果

函数头如下：

```
template<typename T>
T Matrix<T>::dotmul(int r1, T* nums1, int r2, T* nums2)
```

ROI实现的分析

ROI相关操作的初步实现(Hard Copy)

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

对于一个子矩阵而言，重要的信息有子矩阵起始的位置，子矩阵的行数以及列数。故对于一个矩阵而言应该有一些方法可以用来调整这些参数，以控制ROI 的信息。

在本程序当中实现了改变ROI 起始坐标以及行数列数的基本方法，以及返回一个ROI区域的方法。

一开始返回ROI 区域是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。

核心代码如下所示：

```
    int pos =0;
    for(int i = ROIBeginRow; i < ROIBeginRow + ROIRow; i++){
        for(int j = ROIBeginCol; j < ROIBeginCol + ROICol; j ++){
            for (int k = 1; k <= channel; k++) {
                mat.nums[pos++] = nums[k + j * channel + i * ROICol
* channel];
            }
        }
    }
```

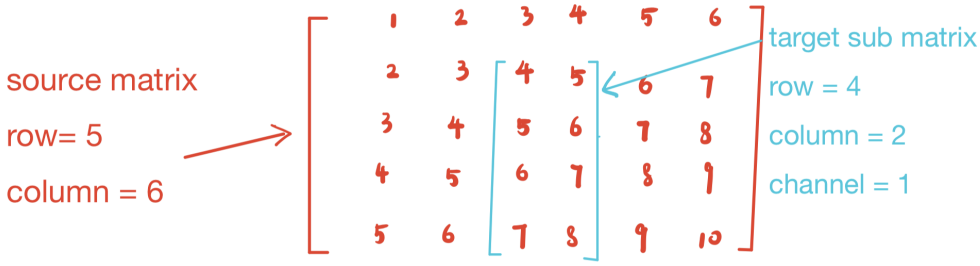
在函数的最后返回生成的子矩阵。

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，所以考虑使用返回子矩阵引用的方式实现ROI操作，为此，进行了一组获取不同大小ROI矩阵的时间效率对比试验，具体结果可以见第三部分。

下面接着讨论如何实现在不进行Hard Copy 的条件下完成对子矩阵的提取。

在母矩阵当中子矩阵元素的提取公式推导

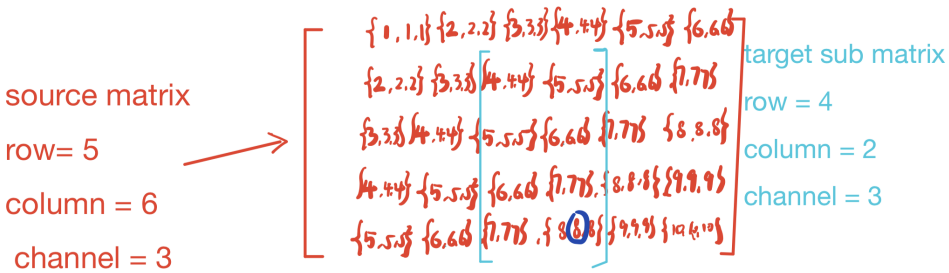
首先考虑单通道矩阵的子矩阵提取的情况（如下图所示）：



由于在矩阵类当中，矩阵的元素数组是通过一段连续的一维数组来表示的，所以在获取子矩阵的（2，1，1）元素，即行数为2， 列数为1， 通道数为1的元素的时候，对于起始位置为子矩阵的头地址时，可以推导出使用公式

$$2 * srcColumn * srcChannel + 1 * srcChannel + 1 - 1$$

来进行计算，可以准确地获取到元素7。那么对于多通道的矩阵，该公式是否依然成立呢？下面进行进一步验证：



如果要获取目标元素（3， 1， 2），以子矩阵头地址为起始点，使用上述公式

$$3 * srcColumn * srcChannel + 1 * srcChannel + 2 - 1$$

计算得到的值为58，经过验证，返回的恰好为深蓝色圈所框起来的元素8，故该公式正确。

经过推导，可以重新实现ROI获取子矩阵的方法，即只要将头地址赋值给子矩阵的元素数组的首地址，并将子矩阵的跨度成员变量Span设置为源矩阵的列数乘以源矩阵的通道数即可，由于子矩阵的通道数与源矩阵的通道数一致，所以直接赋值即可。

重新实现之后，代码避免了逐一复制元素，只是将对应的头地址赋值给子矩阵的元素数组指针即可，核心代码如下所示：

```
    Matrix mat;
    mat.row = ROIRow;
    mat.col = ROICol;
    mat.channel = channel;

    mat.nums = this -> nums + ROIBeginRow * col * channel +
ROIBeginCol * channel;
    mat -> ref_count = this -> ref_count;
    mat -> ref_count ++;
    mat -> span = this -> col * this -> channel;//将子矩阵的跨度变量
设置为母矩阵的列数乘以通道数
    return mat;
```

Part2 Code

```
#ifndef _TEST_MATRIX_H
#define _TEST_MATRIX_H
#include<iostream>
#include<cstring>
#include<fstream>
#include<sstream>


template<class T>
class Matrix{
private:
    int row;
    int col;
```

```
    int channel;
    T *nums;
    int ROIBeginRow;
    int ROIBeginCol;
    int ROIRow;
    int ROICol;
    int span;
    static int num_matrices;
    int* ref_count;
public:

    //返回矩阵对象存在个数
    static int Matrix_num();
    //返回共用该元素数组矩阵对象个数
    int get_ref_count(){
        return ref_count[0];
    }

    int getRow(){
        return row;
    }
    int getCol(){
        return col;
    }
    int getChannel(){
        return channel;
    }
    int getSpan(){
        return span;
    }

    //构造器
    //根据传入数组来创建矩阵的构造函数
    Matrix(int row = 3, int col = 3, int channel = 1, T* nums =
nullptr, int ROIBeginRow = 0, int ROIBeginCol = 0, int ROIRow = 0,
int ROICol = 0, int *ref_count = nullptr, int span = 0);
    //根据传入文件流来创建矩阵的构造函数
    Matrix(int row, int col, int channel, std::ifstream & fin, int
size);

    //向量点乘
    static T dotmul(int r1, T* nums1, int r2, T* nums2);

    //复制构造器
    Matrix(const Matrix&);
    //析构函数
    ~Matrix();
```

```
//矩阵加法运算符
Matrix operator+(T num) const;
Matrix operator+(const Matrix &mat)const ;
Matrix& operator+=(const Matrix &mat) ;
Matrix& operator+=(T num);
//友元函数
template<class TT>
friend Matrix operator+(TT num, const Matrix<TT>& mat){
    return mat + num;
}

//矩阵减法运算符
Matrix operator-(T num) const ;
Matrix operator-(const Matrix &mat)const ;

Matrix& operator-=(T num);
Matrix& operator-=(const Matrix &mat);

template<class TT>
friend Matrix operator-(TT num, const Matrix<TT>& mat){
    return mat - num;
}

//矩阵除法运算符
Matrix operator/(T num) const ;
Matrix& operator/=(T num);

//矩阵乘法运算符
Matrix operator*(T num) const;
Matrix operator*(Matrix &mat);
Matrix& operator*=(T num);

template<class TT>
friend Matrix operator*(TT num, const Matrix<TT>& mat){
    return mat * num;
}

//矩阵转置运算符
Matrix operator!() const;

//自增运算符
//前缀
Matrix& operator++();
//后缀
Matrix operator++(int);
//自减运算符
```

```
    //前缀
    Matrix& operator--();
    //后缀
    Matrix operator--(int);

    //重载赋值运算符
    Matrix& operator=(const Matrix & mat);

    //重载<<运算符
    template<class TT>
    friend std::ostream & operator<<(std::ostream &os, Matrix<TT>&
mat);

    //矩阵元素访问运算符
    T& operator()(int r, int c, int ch);

    //相等运算符
    bool operator==(Matrix& mat);

    //返回矩阵里面元素个数
    int size(){
        return row * col;
    }
    //判断矩阵是否为空
    bool empty(){
        return size() == 0;
    }

    //返回矩阵通道个数
    int channel_num(){
        return channel;
    }

    //ROI相关操作
    //返回子矩阵
    Matrix ROI()const;
    //调整子矩阵位置
    void setROI(int row, int col, int ROIRow, int ROICol);
    void setROIPosition(int row, int col);
    void setROISize(int r, int c);

    //向矩阵尾部添加若干行数据
    Matrix& append(T *nums, int r, int c, int ch);
    //向矩阵尾部删除若干行数据
    Matrix& subtract(int r);
    //矩阵合并(纵向)
    Matrix merge_vertical(const Matrix& mat) const;
```

```
    //矩阵合并（横向）
    Matrix merge_horizontal(const Matrix& mat) const;

};

//将静态类成员初始化
template<class T>
int Matrix<T>::num_matrices = 0;

//返回当前矩阵对象的个数
template<typename T>
int Matrix<T>::Matrix_num(){
    return num_matrices;
}

//默认构造器
template<typename T>
Matrix<T>::Matrix(int row, int col, int channel, T* nums, int
ROIBeginRow, int ROIBeginCol, int ROIRow, int ROICol, int*
ref_count, int span ){
    /*输出检查传入参数
    printf("channel = %d, row = %d, col = %d\nThe element array is
\n", channel, row, col);
    for(int i = 0; i < row * col * channel; i++){
        std::cout << nums[i] << " ";
    }
    std::cout << std::endl;*/
    // std::cout << "Default constructor is invoked.\n";
    //参数检查合法性
    if(row < 0 || col < 0 || ROIBeginCol < 0 || ROIBeginRow < 0 ||
ROIRow < 0 || ROICol < 0){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The input of rows and columns should not less
than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(channel < 1 || channel > 3){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The input of channel should between 1 and 3"
<< std::endl;
        exit(EXIT_FAILURE);
    }
    //如果该矩阵的元素数组没有被引用过，则初始化为1，否则将引用指针指向
传入指针
    if(ref_count == nullptr){
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
```

```
    }else{
        this -> ref_count = ref_count;
    }

    //对ROI的边界的合法性进行检查
    if(ROIBeginRow + ROIRow > row || ROIBeginCol + ROICol > col){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "ROIBeginRow = " << ROIBeginRow << " and
ROIRow = " << ROIRow << " and row = " << row << std::endl;
        std::cerr << "ROIBeginCol = " << ROIBeginCol << " and
ROICol = " << ROICol << " and col = " << col << std::endl;
        std::cerr << "The region of interest should not exceed the
region of the matrix." << std::endl;
        exit(EXIT_FAILURE);
    }

    //成员赋值
    this -> row = row;
    this -> col = col;
    this -> channel = channel;
    this -> ROIBeginRow = ROIBeginRow;
    this -> ROIBeginCol = ROIBeginCol;
    this -> ROIRow = ROIRow;
    this -> ROICol = ROICol;
    if(span == 0){
        this -> span = channel * col;
    }else{
        this -> span = span;
    }

    //根据通道数创建元素数组
    if(nums == nullptr){
        this -> nums = new T[row * col * channel];
    }else{
        int ele_num = row * col * channel;
        this -> nums = new T[row * col * channel];
        for(int i = 0; i < ele_num; i++){
            this -> nums[i] = nums[i];
        }
    }
    num_matrices ++;
    // std::cout << num_matrices << " member exist \n";

}

template<typename T>
```

```
Matrix<T>::Matrix(int row, int col, int channel, std::ifstream &
fin, int size){

    //参数正确性检查
    if(row < 0 || col < 0){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The input of rows and columns should not less
than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(channel < 1 || channel > 3){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The input of channel should between 1 and 3"
<< std::endl;
        exit(EXIT_FAILURE);
    }
    this -> ref_count = new int[1];
    this -> ref_count[0] = 1;

    //对于传入的文件当中包含的数组大小进行检查
    if(row * col * channel != size){
        std::cerr << "In default constructor" << std::endl;
        std::cerr << "The row = " << row << " and the column = " <<
col << " and the channel = " << channel << std::endl;
        std::cerr << "The input array's size should equal to the
size of matrix's element array." << std::endl;
        exit(EXIT_FAILURE);
    }



    //成员赋值
    this -> row = row;
    this -> col = col;
    this -> channel = channel;
    this -> ROIBeginRow = 0;
    this -> ROIBeginCol = 0;
    this -> ROIRow = 0;
    this -> ROICol = 0;
    this -> span = col * channel;

    this -> nums = new T[size];
    int pos = 0;

    while(!fin.eof()){
        std::string line;
        getline(fin, line);
```

```
            std::stringstream ss(line);
            T f;
            while(ss >> f){
                this -> nums[pos++] = f;
            }
        }
        // printf("out\n");
        num_matrices ++;

}


//复制构造函数
template<typename T>
Matrix<T>::Matrix(const Matrix<T>& mat){
    // std::cout << "Copy constructor is invoked.\n";

    this -> row = mat.row;
    this -> col = mat.col;
    this -> channel = mat.channel;
    this -> ROIBeginRow = mat.ROIBeginRow;
    this -> ROIBeginCol = mat.ROIBeginCol;
    this -> ROIRow = mat.ROIRow;
    this -> ROICol = mat.ROICol;
    this -> ref_count = mat.ref_count;
    this -> span = mat.span;

    this -> ref_count[0] ++;
    this -> nums = mat.nums;

    num_matrices ++;
    // std::cout << num_matrices << " members exist\n";
    // std::cout << ref_count[0] << " matrices share the element
array.\n";

}

//析构函数
template<typename T>
Matrix<T>::~Matrix(){
    // printf("Destructor is invoked.\n");
    num_matrices--;
    // std::cout << num_matrices << " member exist\n";

    if(this -> ref_count[0] == 1){
        // printf("In ref_count\n");
        delete[] this -> ref_count;
```

```
        this -> ref_count = nullptr;
        // printf("After deleting ref_count pointer.\n");
        delete[] this -> nums;
        this -> nums = nullptr;
        // std::cout << "The element array has been free.\n";
    }

    else{
        this -> ref_count[0]--;
        // std::cout << this -> ref_count[0] << " matrices share
the element array.\n";
    }

}

//重载赋值运算符
template<typename T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T> & mat){
    // printf("Assignment operator is invoked.\n");
    //参数检查

    //防止自反赋值
    if(this == &mat){
        return *this;
    }
    //如果不是自反赋值，则释放原有对象成员指针指向的内存并将非指针变量一
一赋值
    if(this -> ref_count[0] == 1){
        delete[] nums;
        // std::cout << "The original element array has been
free.\n";
    }else{
        this -> ref_count[0] --;
    }

    this -> row = mat.row;
    this -> col = mat.col;
    this -> channel = mat.channel;
    this -> ROIBeginRow = mat.ROIBeginRow;
    this -> ROIBeginCol = mat.ROIBeginCol;
    this -> ROIRow = mat.ROIRow;
    this -> ROICol = mat.ROICol;
    this -> span = mat.span;

    this -> ref_count = mat.ref_count;

    this -> ref_count[0] ++;
```

```
    // std::cout << this -> ref_count[0] << " matrices share the
element array.\n";

    this -> nums = mat.nums;

    return *this;

}

//加法运算符重载
template <typename T>
Matrix<T> Matrix<T>::operator+(T num) const{
    Matrix<T> mat;
    mat.row = row;
    mat.col = col;
    mat.channel = channel;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.span = span;
    mat.nums = new T[row * col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
this -> nums[i * this -> span + j * this -> channel + ch -1] + num;
                // mat(i, j, ch) = (*this)(i, j, ch) + num;
            }
        }
    }
    return mat;
}

template<typename T>
Matrix<T> Matrix<T>::operator+(const Matrix<T> &m)const{

    //参数合法性检查
    if(row != m.row || col != m.col || channel != m.channel){
        std::cerr << "In '+' operator overloading..." << std::endl;
        std::cerr << "The size of the two matrix should be the
same." << std::endl;
        exit(EXIT_FAILURE);
    }

    Matrix<T> mat;
```

```
    mat.row = row;
    mat.col = col;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.channel = channel;
    mat.span = span;

    int ele_num = row * col * channel;
    mat.nums = new T[ele_num];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
                this -> nums[i * this -> span + j * this -> channel
+ ch - 1]
                + m.nums[i * m.span + j * m.channel + ch - 1];
            }
        }
    }
    return mat;
}

template <typename T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T> &mat){
    //参数检查
    if(row != mat.row || col != mat.col || channel != mat.channel){
        std::cerr << "In assignment operator overloading..." <<
std::endl;
        std::cerr << "The number of row and col and channel of the
two matrix should be the same." << std::endl;
        exit(EXIT_FAILURE);
    }

    int ele_num = row * col * channel;//总的元素个数
    if(this -> ref_count[0] == 1){//如果引用次数为1，则直接改变原数组
的元素值

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
```

```
                    + mat.nums[i * mat.span + j * mat.channel + ch
- 1];
                }
            }
        }
    }else{//如果引用数不为1，则需要重新开辟一块新的空间用于存储新的元素
值
        this -> ref_count[0] --;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];//开辟一块新的空间存储元素数组
        int pos = 0;

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                + mat.nums[i * mat.span + j * mat.channel + ch -
1];
                }
            }
        }
        this -> nums = n;
    }
    return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::operator+=(T num){
    int ele_num = row * col * channel;
    if(this -> ref_count[0] == 1){
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    + num;
                }
            }
        }
    }else{
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
```

```
        T* n = new T[ele_num];
        int pos = 0;
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                    + num;
                }
            }
        }
        this -> nums = n;
    }

    return *this;
}

//减法运算符重载
template <typename T>
Matrix<T> Matrix<T>::operator-(T num) const{
    Matrix<T> mat;
    mat.row = row;
    mat.col = col;
    mat.channel = channel;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.span = span;
    mat.nums = new T[row * col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
this -> nums[i * this -> span + j * this -> channel + ch -1] - num;

            }
        }
    }
    return mat;
}

template<typename T>
Matrix<T> Matrix<T>::operator-(const Matrix<T> &m)const{

    //参数合法性检查
```

```
    if(row != m.row || col != m.col || channel != m.channel){
        std::cerr << "In '+' operator overloading..." << std::endl;
        std::cerr << "The size of the two matrix should be the
same." << std::endl;
        exit(EXIT_FAILURE);
    }

    Matrix<T> mat;
    mat.row = row;
    mat.col = col;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.channel = channel;
    mat.span = span;

    int ele_num = row * col * channel;
    mat.nums = new T[ele_num];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
                this -> nums[i * this -> span + j * this -> channel
+ ch - 1]
                - m.nums[i * m.span + j * m.channel + ch - 1];
            }
        }
    }
    return mat;
}

template <typename T>
Matrix<T>& Matrix<T>::operator-=(const Matrix<T> &mat){
     //参数检查
    if(row != mat.row || col != mat.col || channel != mat.channel){
        std::cerr << "In assignment operator overloading..." <<
std::endl;
        std::cerr << "The number of row and col and channel of the
two matrix should be the same." << std::endl;
        exit(EXIT_FAILURE);
    }

    int ele_num = row * col * channel;//总的元素个数
    if(this -> ref_count[0] == 1){//如果引用次数为1，则直接改变原数组
的元素值
```

```
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    - mat.nums[i * mat.span + j * mat.channel + ch
- 1];
                }
            }
        }
    }else{//如果引用数不为1，则需要重新开辟一块新的空间用于存储新的元素
值
        this -> ref_count[0] --;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];//开辟一块新的空间存储元素数组
        int pos = 0;

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                - mat.nums[i * mat.span + j * mat.channel + ch -
1];
                }
            }
        }
        this -> nums = n;
    }
    return *this;
}

template<typename T>
Matrix<T>& Matrix<T>::operator-=(T num){
    int ele_num = row * col * channel;
    if(this -> ref_count[0] == 1){
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
```

```
                    - num;
                }
            }
        }
    }else{
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];
        int pos = 0;
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                    - num;
                }
            }
        }
        this -> nums = n;
    }

    return *this;
}

//除法运算符重载
template <typename T>
Matrix<T> Matrix<T>::operator/(T num) const{
    Matrix<T> mat;
    mat.row = row;
    mat.col = col;
    mat.channel = channel;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.span = span;
    mat.nums = new T[row * col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
this -> nums[i * this -> span + j * this -> channel + ch -1] / num;

            }
        }
```

```
    }
    return mat;
}

template<typename T>
Matrix<T>& Matrix<T>::operator/=(T num){
    int ele_num = row * col * channel;
    if(this -> ref_count[0] == 1){
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    / num;
                }
            }
        }
    }else{
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];
        int pos = 0;
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                    / num;
                }
            }
        }
        this -> nums = n;
    }

    return *this;
}

//乘法运算符重载
template <typename T>
Matrix<T> Matrix<T>::operator*(T num)const{
    Matrix mat;
    mat.row = row;
    mat.col = col;
    mat.channel = channel;
```

```
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.span = span;
    mat.nums = new T[row * col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] =
this -> nums[i * this -> span + j * this -> channel + ch -1] * num;
            }
        }
    }
    num_matrices++;
    return mat;
}
template<typename T>
Matrix<T>& Matrix<T>::operator*=(T num){
    int ele_num = row * col * channel;
    if(this -> ref_count[0] == 1){
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    * num;
                }
            }
        }
    }else{
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];
        int pos = 0;
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                    * num;
                }
            }
```

```
        }
        this -> nums = n;
    }

    return *this;
}

//矩阵乘法
template<typename T>
Matrix<T> Matrix<T>::operator*(Matrix<T> &mat){

    //判断参数合法性
    if(mat.row != this -> col || mat.channel != channel){
        std::cerr << "In operator * friend functoin..." <<
std::endl;
        std::cerr << "The two matrices' sizes and channel number
should be the same." << std::endl;
        exit(EXIT_FAILURE);
    }

    Matrix<T> res;
    num_matrices++;
    res.row = row;
    res.col = mat.col;
    res.channel = channel;
    res.ROIBeginRow = 0;
    res.ROIBeginCol = 0;
    res.ROIRow = 0;
    res.ROICol = 0;
    res.span = res.channel * res.col;
    int ele_num = res.row * res.col * res.channel;
    res.nums = new T[ele_num];
    /*
    for(int ch = 1; ch <= channel; ch ++){//不同通道数的切换
         for(int i = 0; i < row; i ++){
            for(int j = 0; j < mat.col; j ++){
                for(int k = 0; k < mat.row; k ++){
                    res(i, j, ch) += (*this)(i, k, ch) * mat(k, j,
ch);
                }
            }
        }
    }*/
    /*
    //不同通道数的切换
    for(int i = 0; i < row; i ++){
        for(int j = 0; j < mat.col; j ++){
```

```
            for(int k = 0; k < mat.row; k ++){
                for(int ch = 1; ch <= channel; ch ++){
                    res(i, j, ch) += (*this)(i, k, ch) * mat(k, j,
ch);
                }
            }
        }
    }
    */

    /**/
    //不同通道数的切换
    for(int k = 0; k < mat.row; k ++){
        for(int i = 0; i < row; i ++){
            for(int j = 0; j < mat.col; j ++){
                for(int ch = 1; ch <= channel; ch ++){
                    res(i, j, ch) += (*this)(i, k, ch) * mat(k, j,
ch);
                }
            }
        }
    }

    return res;
}


//矩阵转置运算符
template<typename T>
Matrix<T> Matrix<T>::operator!() const {
    Matrix<T> mat;
    mat.row = this -> col;
    mat.col = this -> row;
    mat.ROIBeginRow = this -> ROIBeginCol;
    mat.ROIBeginCol = this -> ROIBeginRow;
    mat.ROIRow = this -> ROICol;
    mat.ROICol = this -> ROIRow;
    mat.channel = this -> channel;
    int ele_num = mat.row * mat.col * mat.channel;
    mat.nums = new T[ele_num];
    mat.span = mat.col * mat.channel;


    for(int i = 0; i < mat.row; i++){
        for(int j = 0; j < mat.col; j++){
            for(int k = 1; k <= mat.channel; k++){
```

```
                mat(i, j, k) = this -> nums[j * this -> span + i *
this -> channel + k - 1];
            }
        }
    }
    return mat;
}

//前缀自增运算符
template<typename T>
Matrix<T>& Matrix<T>::operator++(){
    int ele_num = row * col * channel;//总的元素个数
    if(this -> ref_count[0] == 1){//如果引用次数为1，则直接改变原数组
的元素值

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    + 1;
                }
            }
        }
    }else{//如果引用数不为1，则需要重新开辟一块新的空间用于存储新的元素
值
        this -> ref_count[0] --;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];//开辟一块新的空间存储元素数组
        int pos = 0;

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j *
this -> channel + ch - 1]
                    + 1;
                }
            }
        }
        this -> nums = n;
    }
    return *this;
}
```

```
//后缀自增运算符
template<typename T>
Matrix<T> Matrix<T>::operator++(int){
    Matrix<T> old = *this; // 保持初始值
    operator++();  // 前缀自增
    return old; //返回初始值
}

//前缀自减运算符
template<typename T>
Matrix<T>& Matrix<T>::operator--(){
    int ele_num = row * col * channel;//总的元素个数
    if(this -> ref_count[0] == 1){//如果引用次数为1，则直接改变原数组的元素值

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this -> channel + ch - 1]
                    = this -> nums[i * this -> span + j * this -> channel + ch - 1]
                    - 1;
                }
            }
        }
    }else{//如果引用数不为1，则需要重新开辟一块新的空间用于存储新的元素值
        this -> ref_count[0] --;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];//开辟一块新的空间存储元素数组
        int pos = 0;

        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j * this -> channel + ch - 1]
                    - 1;
                }
            }
        }
        this -> nums = n;
    }
    return *this;
}
```

```
//后缀自减运算符
template<typename T>
Matrix<T> Matrix<T>::operator--(int){
    Matrix<T> old = *this; // 保持初始值
    operator--();  // 前缀自增
    return old; //返回初始值
}
//矩阵元素访问运算符
template<typename T>
T & Matrix<T>::operator()(int r, int c, int ch){
    //参数合法性检查
    if(r < 0 || c < 0){
        std::cerr << "The input of row and column should not be
less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(ch < 1 || ch > channel){
        std::cerr << "The input of channel should be between 1 and
3" << std::endl;
        exit(EXIT_FAILURE);
    }

    return nums[r * span + c * channel + ch - 1];
}


//相等运算符重载
template<typename T>
bool Matrix<T>::operator==(Matrix<T>& mat){

    if(row != mat.row || col != mat.col || ROIBeginRow !=
mat.ROIBeginRow || ROIBeginCol != mat.ROIBeginCol
    || ROIRow != mat.ROIRow || ROICol != mat.ROIRow || channel !=
mat.channel){
        return false;
    }


    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){

                if((i, j, ch) != mat(i, j, ch)) {
                    return false;
                }
            }
        }
```

```
    }

    return true;

}

//重载<<运算符
template<class T>
std::ostream & operator<<(std::ostream &os, Matrix<T>& mat){
    int ch = mat.channel;
    int r = mat.row;
    int c = mat.col;
    os << "[";
    for(int i = 0; i < r; i++){
        for(int j = 0; j < c; j++){
            os << "{";
            for(int k = 1; k <= ch; k++){
                os << mat(i, j, k);
                if(k != ch)  os << " ";
            }
            os << "}";
            if(j != c - 1) os << ", ";
        }
        if(i != r - 1) os << "\n";
    }
    os << "]";
    return os;
}

//ROI相关操作
//返回子矩阵
template<typename T>
Matrix<T> Matrix<T>::ROI() const{
    //对ROI的边界的合法性进行检查
    if(ROIBeginRow + ROIRow > row || ROIBeginCol + ROICol > col){
        std::cerr << "The region of interest should not exceed the
region of the matrix." << std::endl;
        exit(EXIT_FAILURE);
    }
    Matrix mat;
    num_matrices++;
    mat.row = ROIRow;
    mat.col = ROICol;
    mat.channel = channel;
    /*
    mat.nums = this -> nums + ROIBeginRow * col * channel +
ROIBeginCol * channel;
```

```
    mat.ref_count = this -> ref_count;
    mat.ref_count[0] ++;*/

    int pos =0;
    int ele_num = mat.row * mat.col * mat.channel;
    mat.nums = new T[ele_num];
    for(int i = ROIBeginRow; i < ROIBeginRow + ROIRow; i++){
        for(int j = ROIBeginCol; j < ROIBeginCol + ROICol; j ++){
            for (int k = 1; k <= channel; k++) {
                mat.nums[pos++] = nums[i * ROICol * channel + j *
channel + k - 1];
            }
        }
    }
    // std::cout << mat.ref_count[0] << " matrix share the array
space." << std::endl;
    mat.span = this -> col * this -> channel;
    std::cout << "Finish creating sub matrix." << std::endl;
    return mat;

}

template<typename T>
void Matrix<T>::setROI(int row, int col, int ROIRow, int ROICol){
    //参数正确性检查
    if(row < 0 || col < 0 || ROIRow < 0 || ROICol < 0){
        std::cerr << "In function setROI..." << std::endl;
        std::cerr << "The input of row and column should not be
less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }

    this -> ROIBeginCol = col;
    this -> ROIBeginRow = row;
    this -> ROIRow = ROIRow;
    this -> ROICol = ROICol;
}

template<typename T>
void Matrix<T>::setROIPosition(int row, int col){
    //参数正确性检查
    if(row < 0 || col < 0){
        std::cerr << "In function setROIPosition..." << std::endl;
        std::cerr << "The input of row and column should not be
less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
```

```
    if(row > this -> row || col > this -> col){
        std::cerr << "In function setROIPosition..." << std::endl;
        std::cerr << "The input of row and column should not be
larger than the row number or colomn number of the source matrix."
<< std::endl;
        exit(EXIT_FAILURE);
    }
    //赋值
    this -> ROIBeginRow = row;
    this -> ROIBeginCol = col;
}
template<typename T>
void Matrix<T>::setROISize(int r, int c){
    //参数正确性检查
    if(r < 0 || c < 0){
        std::cerr << "In function setROISize..." << std::endl;
        std::cerr << "The input of row and column should not be
less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }

    this -> ROIRow = r;
    this -> ROICol = c;
};

//向矩阵尾部添加若干行数据
template<typename T>
Matrix<T>& Matrix<T>::append(T *nums, int r, int c, int ch){

    //参数合法性检查
    if(nums == NULL){
        std::cerr << "In function append..." << std::endl;
        std::cerr << "The input array should be valid." <<
std::endl;
        exit(EXIT_FAILURE);
    }
    if(r <= 0 || c <= 0){
        std::cerr << "In function append..." << std::endl;
        std::cerr << "The input of row and column should not be
less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(c != col || ch != channel){
        std::cerr << "In function append..." << std::endl;
        std::cerr << "The channel and column should equal to the
source matrix's channel and column." << std::endl;
        exit(EXIT_FAILURE);
```

```
    }


    int src_ele_num = row * col * channel;//原矩阵元素个数
    int tar_ele_num = r * c * ch;//新传入元素数组的大小数组的大小
    T * arr = new T[src_ele_num + tar_ele_num];//创建新元素数组，大小为两个数组大小之和
    int pos = 0;
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                arr[pos++] = (*this)(i, j, ch);
            }
        }
    }
    for(int j = 0; j < tar_ele_num; j++){
        arr[pos++] = nums[j];
        //在新数组后方赋值为新传入的数组
    }
    if(this -> ref_count[0] == 1){
        delete[] this -> nums;//防止内存泄漏，释放源矩阵原有的元素数组所占空间
    }else{//将原有引用值加一，并将新的引用值设置为1
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
    }

    this -> nums = arr;//将矩阵对象的元素数组指向新开辟的内存区域
    this -> row = this -> row + r;
    return *this;

}

//向矩阵尾部删除若干行数据
template<typename T>
Matrix<T>& Matrix<T>::subtract(int r){
    if(r < 0){
        std::cerr << "In function subtract..." << std::endl;
        std::cerr << "The input of delete row should not be less than 0." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(r > this -> row){
        std::cerr << "In function subtract..." << std::endl;
        std::cerr << "The input of delete row should not be larger than the source matrix's row number." << std::endl;
```

```
        exit(EXIT_FAILURE);
    }

    int ele_num = (row - r) * col * channel;//新数组大小
    T* arr = new T[ele_num];
    int pos = 0;
    for(int i = 0; i < row - r; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                arr[pos++] = (*this)(i, j, ch);
            }
        }
    }
    if(this -> ref_count[0] == 1){
        delete[] this -> nums;//防止内存泄漏，释放源矩阵原有的元素数组所占空间
    }else{//将原有引用值加一，并将新的引用值设置为1
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
    }
    this -> nums = arr;//赋值
    this -> row -= r;
    return *this;
}

//矩阵合并(纵向)
template <typename T>
Matrix<T> Matrix<T>::merge_vertical(const Matrix<T>& mat) const {
    //检查参数合法性

    //传入矩阵成员数组为空的不合法情况
    if(mat.nums == nullptr){
        std::cerr << "In function merge_vertical..." << std::endl;
        std::cerr << "The elements array of source matrix should be valid" << std::endl;
        exit(EXIT_FAILURE);
    }
    //两个矩阵规模或者大小不一致的情况
    if(this -> col != mat.col || this -> channel != mat.channel){
        std::cerr << "In function merge_vertical..." << std::endl;
        std::cerr << "The column and channel of the two matrix should be the same." << std::endl;
        exit(EXIT_FAILURE);
    }

    int row1 = row;
```

```
    int row2 = mat.row;

    Matrix<T> res;
    num_matrices++;
    res.row = row1 + row2;
    res.col = col;
    res.channel = channel;
    res.nums = new T[res.row * col * channel];
    res.span = res.col * res.channel;

    int pos = 0;
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                res(i, j, ch) = this -> nums[i * this -> span + j * this -> channel + ch - 1];
                // res(i, j, ch) = this -> nums[i * this -> span + j * this -> channel + ch - 1];
            }
        }
    }


    for(int i = 0; i < mat.row; i++){
        for(int j = 0; j < mat.col; j++){
            for(int ch = 1; ch <= mat.channel; ch++){
                res(i + row1, j, ch) = mat.nums[i * mat.span + j * mat.channel + ch - 1];
            }
        }
    }

    return res;

}
//矩阵的合并(横向)
template<typename T>
Matrix<T> Matrix<T>::merge_horizontal(const Matrix<T> & mat)const {
    //检查参数合法性

    //传入矩阵成员数组为空的不合法情况
    if(mat.nums == nullptr){
        std::cerr << "In function merge_vertical..." << std::endl;
        std::cerr << "The elements array of source matrix should be valid" << std::endl;
        exit(EXIT_FAILURE);
    }
```

```
    //两个矩阵规模或者大小不一致的情况
    if(row != mat.row || channel != mat.channel){
        std::cerr << "In function merge_vertical..." << std::endl;
        std::cerr << "The row and channel of the two matrix should
be the same." << std::endl;
        exit(EXIT_FAILURE);
    }
    Matrix<T> res;
    num_matrices++;
    res.row = row;
    res.col = col + mat.col;
    res.channel = channel;
    res.span = res.col * res.channel;
    res.nums = new T[row * res.col * channel];

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col ; j++){
            for(int ch = 1; ch <= channel; ch++){
                // res(i, j, ch) = (*this)(i, j, ch);
                res(i, j, ch) = this -> nums[i * this -> span + j *
this -> channel + ch - 1];
            }
        }

        for(int j = 0; j < mat.col ; j++){
            for(int ch = 1; ch <= channel; ch++){
                // res(i, col + j, ch) = mat(i, j, ch);
                res(i, col + j, ch) = mat.nums[i * mat.span + j *
mat.channel + ch - 1];
            }
        }
    }

    return res;

}


//向量点乘运算
template<typename T>
T Matrix<T>::dotmul(int r1, T* nums1, int r2, T* nums2){

    //参数检查合法性
    if(r1 < 0 || r2 < 0) {
        std::cerr << "Break in dotmul." << std::endl;
        std::cerr << "The input of r1 and r2 should not less than
0." << std::endl;
```

```
        exit(EXIT_FAILURE);
    }
    if(r1 != r2) {
        std::cerr << "Break in dotmul." << std::endl;
        std::cerr << "The input of r1 and r2 should be equivalent."
<< std::endl;
        exit(EXIT_FAILURE);
    }
    //检测空指针
    if(nums1 == nullptr || nums2 == nullptr) {
        std::cerr << "Break in dotmul." << std::endl;
        std::cerr << "The two vectors should be valid." <<
std::endl;
    }
    //矩阵点乘运算
    T result = 0;
    for(int i = 0; i < r1; i++) {
        result += nums1[i] * nums2[i];
    }
    return result;
}

#endif
```

Part3 Result and Verification

各函数正确及可行性验证

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。

先以单通道整型5 * 5 矩阵的创建进行试验：

```
    int *nums = new int[25];
    for(int i = 0; i < 25; i++){
        nums[i] = i+1;
    }

    Matrix<int> mat(5, 5, 1, nums);
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
channel = 1, row = 5, col = 5
The element array is
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

结果可以成功构造出一个矩阵对象

进一步地创建一个浮点数类型的三通道矩阵

```
float *nums = new float[30];
    for(int i = 0; i < 30; i++){
        nums[i] = i+1 + 0.0;
    }

    Matrix<float> mat(2, 5, 3, nums);
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
channel = 3, row = 2, col = 5
The element array is
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

结果依然可以创建一个三通道的浮点数矩阵。且对于不合法的输入会进行错误提示并使得对象创建失败。

```
Matrix<float> mat(2, 5, 3, nums, 1, 2, 4, 2);
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
In default constructor
ROIBeginRow = 1 and ROIRow = 4 and row = 2
ROIBeginCol = 2 and ROICol = 2 and col = 5
The region of interest should not exceed the region of the matrix.
```

向量点乘正确性验证

通过创建两个大小为30的向量nums1 和 nums2，其中num1中的元素为从1到30，nums2中的元素始终为1，通过调用点乘函数实验

```
cout << Matrix<int>::dotmul(30, nums, 30, nums1) << endl;
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
465
```

运算结果正确，之后又进行了对多种数据类型不同大小向量的实验，经过计算器验证，结果均正确。

复制构造函数以及赋值运算符重载的正确性验证

实验思路

通过在默认构造函数中插入语句，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，所以在两个函数当中也分别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。

实验方式

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = i + 1;
    }
    Matrix<int> mat1(5, 6, 1, nums1);
    Matrix<int> mat2(2, 5, 3, nums1);
    Matrix<int> mat3 = mat1;

    Matrix<int> mat4;
    mat4 = mat3;
```

通过默认构造函数显式地创建两个对象mat1和mat2，之后使用复制构造函数创建新的矩阵对象mat3，并将mat3的各元素值赋值为mat1对应的值，其中它们共用一块区域来存储元素数组，这样可以避免硬拷贝，提高创建对象的效率。之后在使用赋值运算符，先创建mat4，之后将mat3的值赋值到mat4的对应元素当中。

实验结果

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
Default constructor is invoked.
1 member exist
Default constructor is invoked.
2 member exist
Copy constructor is invoked.
3 members exist
2 matrices share the element array.
Default constructor is invoked.
4 member exist
Assignment operator is invoked.
The original element array has been free.
3 matrices share the element array.
Destructor is invoked.
3 member exist
2 matrices share the element array.
Destructor is invoked.
2 member exist
1 matrices share the element array.
Destructor is invoked.
1 member exist
The element array has been free.
Destructor is invoked.
0 member exist
The element array has been free.
```

实验分析

首先通过默认构造器创建了两个对象mat1和mat2，创建完成之后，当前的矩阵对象个数是2， 符合逻辑。之后通过复制构造器基于对象mat1创建了对象mat3，此时，mat1和mat3两个矩阵共用一块内存区域存储元素数组的值。之后再使用了默认构造函数创建新的矩阵对象mat4，进一步地使用赋值运算符将mat3的值赋值给mat4，由于mat4原本的元素数组内存区域只有自己在使用，没有别的矩阵与其共享这一块区域，所以直接将该内存区域释放之后，将mat4的内存区域与mat3共享，此时，该内存区域就有3个对象共享。在程序运行完成之后，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，释放之后存在的对象只有3个，由于mat4对象数组所占有的区域还被其它两个矩阵对象共享，所以不会对该区域进行释放。最后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比较成功的。

矩阵访问运算符重载的正确性实验

实验思路

首先将一个长度为30的数组nums作为元素数组通过默认构造函数赋值给三个行数列数以及通道数都不同的三个矩阵对象，三个矩阵的通道数分别为1， 2， 3。赋值完成后，逐行逐列逐通道地使用矩阵访问运算符对他们的元素进行逐个打印，最后确认三个打印的结果是否与数组元素一致，如果一致，则说明矩阵访问运算符重载成功。

实验核心代码

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = i + 1;
    }

    Matrix<int> mat1(5, 6, 1, nums1);
    Matrix<int> mat2(3, 5, 2, nums1);
    Matrix<int> mat3(2, 5, 3, nums1);
    cout << "Matrix 1: \n";
    for(int i = 0; i < mat1.getRow(); i++){
        for(int j = 0; j < mat1.getCol(); j++){
            for(int ch = 1; ch <= mat1.getChannel(); ch++){
                cout << mat1(i, j, ch) << " ";
            }
        }
    }
```

实验结果

```
Matrix 1:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Matrix 2:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Matrix 3:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

经检验，三个矩阵的访问结果均与原数组的值一致，故可验证矩阵访问运算符重载可以正确地访问到矩阵的特定具体值。

矩阵基本运算正确性检验

正如前文所述，矩阵的基本操作包括了加减除法，乘法在下一小节单独阐述，对于实验中以矩阵加法为例，由于对于模板类，传入的参数类型可以是多种数据类型，本实验使用int整型类型，探究上文提到的矩阵加法是否能够正确地计算出结果，尤其是多通道的加法，计算的结果正确性能否保证。进行如下实验：

```
int *nums1 = new int[30];
for(int i = 0; i < 30; i++){
    nums1[i] = i + 1;
}

Matrix<int> mat1(5, 6, 1, nums1);
Matrix<int> mat2(3, 5, 2, nums1);
Matrix<int> mat3 = mat2 + 2;
cout << "mat2: " << endl;
cout <<mat2 << endl;
cout << "mat3: " << endl;
cout << mat3 << endl;
Matrix<int> mat4 = mat3 + mat2;
mat2 += mat3;
cout << "mat2: " << endl;
cout <<mat2 << endl;
cout << "mat4:" << endl;
cout << mat4 << endl;
```

分别将mat2与数字相加赋值给mat3，将mat2 += mat3, 以及将mat2与mat3相乘的结果赋值给mat4，最后通过重载的<<运算符将新创建的两个矩阵的元素打印出来，结果如下图所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ ./a.out
mat2:
[{1 2}, {3 4}, {5 6}, {7 8}, {9 10}
{11 12}, {13 14}, {15 16}, {17 18}, {19 20}
{21 22}, {23 24}, {25 26}, {27 28}, {29 30}]
mat3:
[{3 4}, {5 6}, {7 8}, {9 10}, {11 12}
{13 14}, {15 16}, {17 18}, {19 20}, {21 22}
{23 24}, {25 26}, {27 28}, {29 30}, {31 32}]
mat2:
[{4 6}, {8 10}, {12 14}, {16 18}, {20 22}
{24 26}, {28 30}, {32 34}, {36 38}, {40 42}
{44 46}, {48 50}, {52 54}, {56 58}, {60 62}]
mat4:
[{7 10}, {13 16}, {19 22}, {25 28}, {31 34}
{37 40}, {43 46}, {49 52}, {55 58}, {61 64}
{67 70}, {73 76}, {79 82}, {85 88}, {91 94}]
```

可以发现，运算的结果正确无误。

对于加法友元函数，进行试验

```
mat2 = 2 + mat2;
cout << "mat2: " << endl;
```

打印确认结果依旧正确。所以可以验证矩阵加法的正确性。

对于减法以及除法，由于实现方法与矩阵加法相似，故不多进行试验检验其正确性。只进行两组实验进行验证即可：

```
mat2 /= 2;
cout << "mat2: " << endl;
cout <<mat2 << endl;
mat2 -= 2;
cout << "mat2: " << endl;
cout <<mat2 << endl;
```

```
mat2:
[{1 2}, {3 4}, {5 6}, {7 8}, {9 10}
{11 12}, {13 14}, {15 16}, {17 18}, {19 20}
{21 22}, {23 24}, {25 26}, {27 28}, {29 30}]
mat2:
[{0 1}, {1 2}, {2 3}, {3 4}, {4 5}
{5 6}, {6 7}, {7 8}, {8 9}, {9 10}
{10 11}, {11 12}, {12 13}, {13 14}, {14 15}]
mat2:
[{-2 -1}, {-1 0}, {0 1}, {1 2}, {2 3}
{3 4}, {4 5}, {5 6}, {6 7}, {7 8}
{8 9}, {9 10}, {10 11}, {11 12}, {12 13}]
```

由结果图中可以看出，对于减法以及除法的运算，重载运算符均能保证正确性。

矩阵乘法正确性检验

对于简单的矩阵乘法，例如单通道的两个1 * 3与3 * 1 的小型矩阵互乘，验证单通道乘法的正确性：

```
    Matrix<int> mat1(1, 3, 1, nums2);
    cout << "mat1: \n" <<mat1 << endl;
    Matrix<int> mat2(3, 1, 1, nums2);
    cout <<"mat2: \n" <<mat2 << endl;
    Matrix<int> mat3 = mat2 * mat1;
    cout << "mat3:\n"<<mat3 << endl;
    mat3 = mat1 * mat2;
    cout << "mat3:\n"<<mat3 << endl;
```

产生的结果如下图所示：

可以基本验证单通道矩阵乘法的正确性。下面进行多通道矩阵乘法的正确性验证试验：

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = 1;
    }
    Matrix<int> mat1(2, 5, 3, nums1);
    Matrix<int> mat2(5, 2, 3, nums1);
    cout << "mat1:\n" << mat1 << endl << endl;
    cout << "mat2:\n" << mat2 << endl << endl;

    Matrix<int> mat3 = mat1 * mat2;
    cout << "mat3:\n" << mat3 << endl << endl;
    mat3 = mat2 * mat1;
    cout << "mat3:\n" << mat3 << endl << endl;
```

先是基于一个长度为30的int类型的数组创建两个行列数分别为5， 2和2，5的三通道矩阵。之后在满足乘法合理性的条件下对他们进行先后相乘，将结构分别赋值给mat3矩阵对象。之后对mat3进行打印，为了更加直观地展现矩阵乘法的正确性，对于对象数组使用了全为1的赋值。实验结果如下图所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
mat1:
[{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}]

mat2:
[{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}]

mat3:
[{5 5 5}, {5 5 5}
{5 5 5}, {5 5 5}]

mat3:
[{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}, {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}, {2 2 2}]
```

可以直观地观察到对于两种顺序的多通道乘法结果均为正确。随后又进行了多组不同数据类型的数据量更大的矩阵乘法，经过matlab验证，结果均正确无误。部分实验结果如下图所示：

```
1   6.027915e+04 6.733141e+04 5.663333e+04 6.081126e+04 7.337449e+04 6.670471e+04 7.197914e+04 7.142771e+04 5.736819e
2   6.278781e+04 7.780466e+04 7.642764e+04 7.371781e+04 9.096316e+04 7.412579e+04 9.606066e+04 7.347415e+04 6.672042e
3   5.834378e+04 6.766249e+04 5.732566e+04 5.746363e+04 7.832203e+04 5.799606e+04 7.632863e+04 6.230781e+04 6.334588e
4   6.511606e+04 7.344154e+04 7.033309e+04 6.117042e+04 8.106548e+04 7.229754e+04 7.713734e+04 6.735816e+04 6.949110e
5   7.329750e+04 8.832774e+04 7.679511e+04 8.111910e+04 9.537184e+04 8.573006e+04 9.695482e+04 8.324732e+04 7.636423e
6   7.062764e+04 7.483881e+04 6.782594e+04 7.183101e+04 8.941845e+04 7.755192e+04 8.657121e+04 7.798891e+04 6.460144e
7   7.004144e+04 8.649973e+04 7.413191e+04 7.202921e+04 1.017892e+05 8.103009e+04 9.079456e+04 8.254069e+04 7.724665e
8   7.192127e+04 8.066366e+04 7.584162e+04 6.432426e+04 9.049937e+04 8.023633e+04 8.582382e+04 7.194200e+04 7.494009e
9   5.635195e+04 5.871494e+04 6.012312e+04 5.830336e+04 7.424932e+04 6.082802e+04 7.274847e+04 6.669544e+04 5.926383e
10  7.664699e+04 8.539224e+04 7.411217e+04 7.030413e+04 8.472899e+04 7.796485e+04 8.449738e+04 8.048066e+04 7.037392e
11  6.265836e+04 6.814818e+04 7.290931e+04 5.831881e+04 7.831458e+04 7.680091e+04 8.108149e+04 7.867275e+04 6.012272e
12  5.745684e+04 6.645359e+04 5.835081e+04 5.358140e+04 7.032175e+04 6.714323e+04 6.379397e+04 7.318084e+04 5.703518e
13  6.384671e+04 7.767532e+04 6.977245e+04 5.682827e+04 8.222042e+04 8.182258e+04 7.656428e+04 7.716225e+04 6.706182e
14  5.269816e+04 6.366520e+04 6.349970e+04 4.678028e+04 7.328243e+04 6.972063e+04 6.537416e+04 5.393634e+04 6.099224e
15  7.418509e+04 9.575478e+04 7.093368e+04 8.056383e+04 9.898713e+04 8.914417e+04 9.488534e+04 8.464447e+04 7.300346e
16  7.217975e+04 8.133153e+04 8.156760e+04 6.449258e+04 9.264148e+04 8.779310e+04 9.261817e+04 8.230371e+04 7.383137e
17  7.382315e+04 7.552202e+04 7.649203e+04 6.236193e+04 8.569093e+04 7.802088e+04 8.440828e+04 7.359697e+04 7.542833e
18  8.323174e+04 7.538252e+04 8.916689e+04 6.483667e+04 9.378344e+04 8.890951e+04 9.481572e+04 8.435512e+04 7.952081e
19  6.833908e+04 8.112868e+04 7.174779e+04 5.757929e+04 8.052354e+04 7.843654e+04 7.438646e+04 7.210703e+04 7.009730e
20  6.303053e+04 7.515794e+04 7.069782e+04 5.675626e+04 8.301827e+04 8.142635e+04 6.853669e+04 7.920140e+04 6.404285e
21  7.823854e+04 9.584322e+04 8.661256e+04 8.157708e+04 9.895610e+04 9.390564e+04 9.459214e+04 8.843090e+04 8.737776e
22  7.903626e+04 9.080858e+04 7.926665e+04 7.944322e+04 9.536218e+04 8.314596e+04 8.919400e+04 8.812634e+04 8.150112e
23  7.338336e+04 8.393971e+04 8.295745e+04 7.081791e+04 9.365054e+04 8.741024e+04 8.712773e+04 8.105416e+04 7.704919e
```

```
1.0e+05 *

列 1 至 21

  0.6028   0.6733   0.5663   0.6081   0.7337   0.6670   0.7198   0.7143   0.5737   0.7711   0.7895   0.7674   0.8260   0.8128   0.6969   0.6321   0.6408   0.6722   0.8167   0.5944   0.6611
  0.6279   0.7780   0.7643   0.7372   0.9096   0.7413   0.9606   0.7347   0.6672   0.9285   0.8563   0.8448   0.9364   0.9294   0.8014   0.6839   0.6994   0.7873   0.8562   0.7171   0.7273
  0.5834   0.6766   0.5733   0.5746   0.7832   0.5800   0.7633   0.6231   0.6335   0.7910   0.7550   0.7124   0.7795   0.7463   0.5910   0.6581   0.5760   0.6302   0.7840   0.5905   0.6239
  0.6512   0.7344   0.7033   0.6117   0.8107   0.7230   0.7714   0.6736   0.6949   0.9190   0.8123   0.7558   0.9220   0.8439   0.6660   0.7676   0.6069   0.7472   0.8470   0.6293   0.7190
  0.7330   0.8833   0.7680   0.8112   0.9537   0.8573   0.9695   0.8325   0.7636   0.9625   0.9657   0.8787   0.9489   0.9684   0.8268   0.8119   0.7500   0.9753   0.9745   0.8098   0.7235
  0.7063   0.7484   0.6783   0.7183   0.8942   0.7755   0.8657   0.7799   0.6460   0.8396   0.8024   0.8466   0.8567   0.8613   0.8491   0.8030   0.7157   0.7963   0.8566   0.7627   0.6996
  0.7004   0.8650   0.7413   0.7203   1.0179   0.8103   0.9079   0.8254   0.7725   0.9995   0.9274   0.8322   0.9603   0.8770   0.8160   0.7354   0.8255   0.7903   0.9167   0.7833   0.7564
  0.7192   0.8066   0.7584   0.6432   0.9050   0.8024   0.8582   0.7194   0.7494   1.0093   0.8248   0.7361   0.9446   0.8963   0.7554   0.7430   0.6726   0.6786   0.8820   0.7080   0.6825
  0.5635   0.5871   0.6012   0.5830   0.7425   0.6083   0.7275   0.6670   0.5926   0.6966   0.7070   0.6867   0.7045   0.7166   0.6679   0.6324   0.5793   0.6774   0.7261   0.6638   0.6214
  0.7665   0.8539   0.7411   0.7030   0.8473   0.7796   0.8450   0.8048   0.7037   0.8970   0.9229   0.8499   0.9156   0.9305   0.8112   0.8606   0.6924   0.8947   0.9369   0.7197   0.7463
  0.6266   0.6815   0.7291   0.5832   0.7831   0.7680   0.8108   0.7867   0.6012   0.8132   0.7515   0.8004   0.7523   0.8289   0.8079   0.6700   0.6927   0.6949   0.7600   0.6213   0.7169
  0.5746   0.6645   0.5835   0.5358   0.7032   0.6714   0.6379   0.7318   0.5704   0.7650   0.7404   0.7112   0.7841   0.7850   0.7326   0.5661   0.6195   0.5714   0.7492   0.5664   0.6920
  0.6385   0.7768   0.6977   0.5683   0.8222   0.8182   0.7656   0.7716   0.6706   0.8415   0.8189   0.8279   0.8630   0.8962   0.8020   0.6736   0.8155   0.7376   0.8677   0.6981   0.7055
  0.5270   0.6367   0.6350   0.4678   0.7328   0.6972   0.6537   0.5394   0.6099   0.8226   0.6537   0.7638   0.7574   0.7212   0.6139   0.6238   0.5697   0.6503   0.7429   0.5052   0.5906
  0.7419   0.9575   0.7093   0.8056   0.9899   0.8914   0.9489   0.8464   0.7300   0.9420   0.9187   0.9295   1.0293   0.9675   0.8809   0.9299   0.8423   0.8825   0.9394   0.8094   0.7740
  0.7218   0.8133   0.8157   0.6449   0.9264   0.8779   0.9262   0.8230   0.7383   0.9579   0.8098   0.9064   0.9088   0.9009   0.8251   0.7601   0.7535   0.8874   0.8793   0.7091   0.8103
  0.7382   0.7552   0.7649   0.6236   0.8569   0.7802   0.8441   0.7360   0.7543   0.8885   0.8662   0.8223   0.8883   0.8644   0.8050   0.7821   0.6551   0.7435   0.8747   0.6721   0.7150
  0.8323   0.7538   0.8917   0.6484   0.9378   0.8891   0.9482   0.8436   0.7952   1.0416   0.8692   0.8367   0.9347   0.9564   0.8484   0.7885   0.7611   0.9164   0.9205   0.7382   0.8255
  0.6834   0.8113   0.7175   0.5758   0.8052   0.7844   0.7439   0.7211   0.7010   0.8921   0.8734   0.7959   0.8914   0.9210   0.7835   0.8053   0.7219   0.7710   0.9255   0.7412   0.7001
  0.6303   0.7516   0.7070   0.5676   0.8302   0.8143   0.6854   0.7920   0.6404   0.8198   0.7951   0.7025   0.8445   0.7859   0.8031   0.6926   0.7085   0.6495   0.7315   0.6241   0.6823
  0.7824   0.9584   0.8661   0.8158   0.9896   0.9391   0.9459   0.8843   0.8738   1.1023   1.0689   1.0163   1.0031   0.9898   0.8706   0.8754   0.7411   0.8936   0.9801   0.7294   0.8596
  0.7904   0.9081   0.7927   0.7944   0.9536   0.8315   0.8919   0.8813   0.8150   1.0062   1.0764   0.8695   1.0319   1.0121   0.9009   0.8188   0.8168   0.8151   1.0095   0.7894   0.7735
  0.7338   0.8394   0.8296   0.7082   0.9365   0.8741   0.8713   0.8105   0.7705   1.1040   0.9308   0.8802   0.9756   0.9378   0.8434   0.7277   0.7452   0.7533   0.9344   0.6995   0.7813
  0.6856   0.8632   0.6605   0.6546   0.8148   0.7881   0.8149   0.7610   0.7032   0.8922   0.8924   0.8897   0.8858   0.9292   0.7510   0.7296   0.6970   0.8080   0.9418   0.7138   0.7676
  0.8275   0.9422   0.9017   0.7904   0.9148   0.9343   0.9954   0.9200   0.7179   1.0455   0.9675   0.9274   1.0212   1.0828   0.9656   0.8176   0.7691   0.9138   0.9934   0.7825   0.7709
  0.7149   0.8354   0.6686   0.7958   0.8476   0.7090   0.9298   0.9269   0.6875   0.8824   0.9047   0.8308   0.8999   0.9851   0.9243   0.7285   0.7275   0.7541   0.9268   0.7959   0.7466
  0.7108   0.7858   0.7076   0.6405   0.8481   0.6755   0.8192   0.6678   0.6938   0.8374   0.8928   0.7590   0.8891   0.8583   0.7717   0.8031   0.6226   0.6987   0.8623   0.6751   0.6787
  0.6238   0.7000   0.6806   0.5593   0.7903   0.7330   0.8674   0.7684   0.6564   0.8563   0.8389   0.7346   0.7424   0.8718   0.7513   0.7317   0.6613   0.7000   0.8423   0.6415   0.7105
  0.7533   0.8181   0.8389   0.6120   0.8710   0.8297   0.8700   0.7436   0.6915   1.0003   0.9032   0.8336   0.9778   1.0377   0.8046   0.7216   0.6766   0.8138   0.9396   0.7031   0.7307
  0.7530   1.0285   0.8062   0.7775   1.0174   1.0022   0.9231   0.9518   0.8663   1.1016   1.0183   0.9824   1.0130   1.0452   0.8603   0.8889   0.8786   0.8648   1.0289   0.8584   0.8824
  0.8398   1.0643   0.8463   0.9497   1.1335   0.9802   1.0728   1.0237   0.9246   1.0791   1.1061   1.0178   1.1232   1.0930   0.9834   0.9672   0.9423   0.9142   1.0635   0.9226   0.9441
```

ROI 相关操作正确性检验

本程序的ROI通过返回母矩阵头地址并且将子矩阵的跨度设置为母矩阵的跨度的方式，使得返回的子矩阵不需要额外拷贝母矩阵的大量空间，只需要与母矩阵共用一块元素数组内存区域即可。下面进行正确性的验证：

首先是对于单通道的矩阵进行ROI正确性验证，对于一个行数为6，列数为5的单通道矩阵，通过在创建母矩阵的时候设置ROI的起始行列数以及ROI区域所跨的行数以及列数，可以返回一个与母矩阵共用一块内存区域的子矩阵。

实验代码如下所示：

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = 1 + i;
    }
    Matrix<int> mat1(6, 5, 1, nums1, 2, 1, 2, 3);
    cout << mat1 << endl;
    Matrix<int> sub = mat1.ROI();
    cout << sub << endl;
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
[{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}
{16}, {17}, {18}, {19}, {20}
{21}, {22}, {23}, {24}, {25}
{26}, {27}, {28}, {29}, {30}]

[{12}, {13}, {14}
{17}, {18}, {19}]
```

结果验证了对于单通道的矩阵，ROI提取子矩阵的方法编写是正确的。

接下来进行对于多通道矩阵，ROI提取子矩阵方法的正确性验证：

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = 1 + i;
    }


    Matrix<int> mat1(2, 5, 3, nums1, 0, 1, 2, 3);
    cout << mat1 << endl << endl;
    Matrix<int> sub = mat1.ROI();
    cout << sub << endl;
```

即创建一个行数为2，列数为5， 通道数为3的母矩阵，并从中提取出起始行列数分别为0， 1，行跨度为2， 列跨度为3的子矩阵，当然子矩阵的通道数也为3，最终返回的结果如下图所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]

[{4 5 6}, {7 8 9}, {10 11 12}
{19 20 21}, {22 23 24}, {25 26 27}]
```

下面进行共用内存块以避免硬拷贝以及内存管理有效性的验证：

验证思路与方法

此验证需要在ROI提取子矩阵的同时输出该内存区域同时被多少个矩阵对象所共用，事先先通过复制构造器以及赋值运算符的使用来使得该区域被多个矩阵对象所共用，之后再进行ROI矩阵的提取，之后再次返回有多少个矩阵在共用这一块内存区域，在程序运行结束之后，进一步观察析构函数对于该内存区域回收的处理，这样可以验证该ROI方法避免了使用硬拷贝的低效方式进行子矩阵的提取，进一步也从侧面说明了内存管理的有效性，即析构函数会回收这一块内存区域当且仅当只有该区域只被一个矩阵对象所共用，以此来保证不会出现同一块内存区域被释放两次的情况。

进一步实验代码如下：

```
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = 1 + i;
    }


    Matrix<int> mat1(2, 5, 3, nums1, 0, 1, 2, 3);
    Matrix<int> mat2(mat1);
```

```
    Matrix<int> mat3;
    mat3 = mat1;
    cout << "mat1: \n" << mat1 << endl;
    cout << "mat3:\n" << mat3 << endl << endl;
    Matrix<int> sub = mat1.ROI();
    cout <<"sub:\n" << sub << endl;
```

验证结果以及分析

经过在各构造函数、赋值运算符、析构函数以及返回ROI矩阵的方法当中打印有效信息进行验证，验证结果如下所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ ma
1 member exist
2 members exist
2 matrices share the element array.
3 member exist
The original element array has been free.
3 matrices share the element array.
mat1:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]
mat3:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]

4 member exist
4 matrix share the array space.
sub:
[{4 5 6}, {7 8 9}, {10 11 12}
{19 20 21}, {22 23 24}, {25 26 27}]
Destructor is invoked.
4 member exist
3 matrices share the element array.
Destructor is invoked.
3 member exist
2 matrices share the element array.
```

```
Destructor is invoked.
2 member exist
1 matrices share the element array.
Destructor is invoked.
1 member exist
The element array has been free.
```

分析产生的实验结果，首先创建了三个矩阵mat1``mat2``mat3，其中mat2基于mat1使用复制构造函数得到，mat3先是一个新的矩阵对象，之后通过赋值运算符将所有的值赋值为mat1的值，此时当前程序存在三个矩阵对象，并且三个矩阵对象共享同一块内存区域。之后基于mat1获取创建子矩阵mat4，同样地，在创建子矩阵的过程当中避免了硬拷贝赋值大量内存空间。导致子矩阵sub和另外三个矩阵共享同一块内存区域，此时输出有4个对象共享同一块内存区域，最后当析构函数调用的时候仅当所有的矩阵对

象都被释放，析构函数才会真正地去释放这一块内存区域。所以这也达到了软复制以及保证程序安全性的要求。

在矩阵底部添加或删除元素的正确性检验

单通道矩阵测试

对一个五行六列的通道为1矩阵mat1，在底部添加一个五行六列的数组，之后在底部删除3行元素，最后使用ROI返回子矩阵，实验代码如下：

```
int *nums1 = new int[30];
for(int i = 0; i < 30; i++){
    nums1[i] = 1 + i;
}


Matrix<int> mat1(6, 5, 1, nums1, 0, 1, 2, 3);
cout <<"mat1:\n"<< mat1 << endl << endl;
mat1.append(nums1, 6, 5, 1);
cout << "mat1:\n"<<mat1 << endl << endl;
mat1.subtract(3);
cout << "mat1:\n"<<mat1 << endl << endl;

Matrix<int> sub = mat1.ROI();
cout << "sub:\n"<<sub << endl;
```

运行之后的结果为：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
mat1:
[{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}
{16}, {17}, {18}, {19}, {20}
{21}, {22}, {23}, {24}, {25}
{26}, {27}, {28}, {29}, {30}]

mat1:
[{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}
{16}, {17}, {18}, {19}, {20}
{21}, {22}, {23}, {24}, {25}
{26}, {27}, {28}, {29}, {30}
{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}
{16}, {17}, {18}, {19}, {20}
{21}, {22}, {23}, {24}, {25}
{26}, {27}, {28}, {29}, {30}]
```

```
mat1:
[{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}
{16}, {17}, {18}, {19}, {20}
{21}, {22}, {23}, {24}, {25}
{26}, {27}, {28}, {29}, {30}
{1}, {2}, {3}, {4}, {5}
{6}, {7}, {8}, {9}, {10}
{11}, {12}, {13}, {14}, {15}]

sub:
[{2}, {3}, {4}
{7}, {8}, {9}]
```

故对于单通道的矩阵添加删除元素是可以正确运行的。

多通道矩阵测试

对于创建的新的行数为2，列数为5，通道数为3的矩阵进行测试，测试结果如下所示：

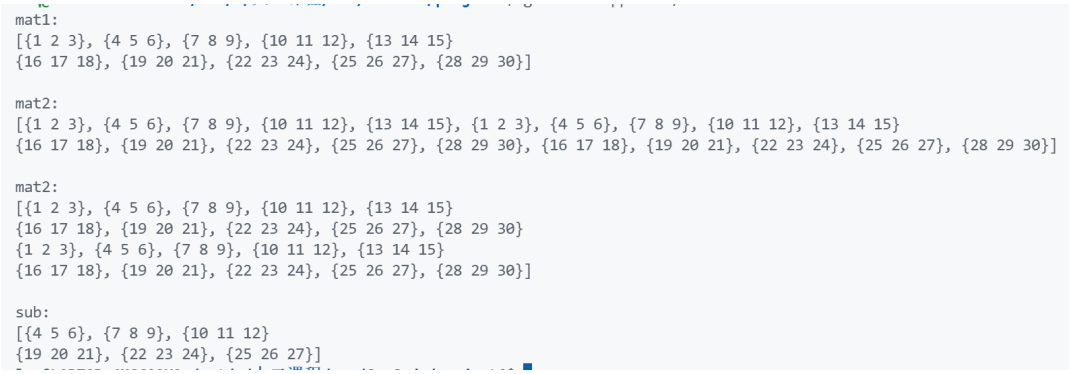
```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
mat1:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]

mat1:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}
{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]

mat1:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}
{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}]

sub:
[{4 5 6}, {7 8 9}, {10 11 12}
{19 20 21}, {22 23 24}, {25 26 27}]
```

可以看出对于多通道的矩阵而言，依然可以正确地在矩阵底部添加删除元素。

矩阵合并正确性验证

同样地，需要验证多通道矩阵mat1对自身做纵向合并与横向合并的正确性，分别将纵向合并与横向合并的结果赋值给mat2

得到的实验结果如下图所示：



对于mat2的第一次输出，输出的是mat1横向合并的结果，第二次输出则是对mat1纵向合并的结果，对于sub矩阵则是输出的对mat1获取子矩阵的结果，由实验结果可以检验矩阵合并方法编写是正确的并且矩阵合并方法不会改变原矩阵的值。

矩阵转置操作正确性验证

有两点需要验证，一是矩阵能否正确进行转置操作，而是矩阵转置之后是否改变了原矩阵的值，在本程序中，矩阵转置是产生一个新的转置之后的矩阵并且不会改变原有矩阵的值。实验代码如下：

```
Matrix<int> mat1(2, 5, 3, nums1, 0, 1, 2, 3);
    Matrix<int> mat3;
    mat3 = !mat1;
    cout << "mat1: \n" << mat1 << endl;
    cout << "mat3:\n" << mat3 << endl;
    Matrix<int> sub = mat1.ROI();
    cout <<"sub:\n" << sub << endl;
```

先是创建源矩阵mat1，之后将对其转置之后的矩阵赋值给mat3，随后打印两个矩阵判断是否成功对矩阵进行转置，之后为了确认原矩阵没有被修改，再额外打印mat1的ROI矩阵。实验结果如下所示：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project4$ g++ main.cpp && ./a.out
mat1:
[{1 2 3}, {4 5 6}, {7 8 9}, {10 11 12}, {13 14 15}
{16 17 18}, {19 20 21}, {22 23 24}, {25 26 27}, {28 29 30}]
mat3:
[{1 2 3}, {16 17 18}
{4 5 6}, {19 20 21}
{7 8 9}, {22 23 24}
{10 11 12}, {25 26 27}
{13 14 15}, {28 29 30}]

sub:
[{4 5 6}, {7 8 9}, {10 11 12}
{19 20 21}, {22 23 24}, {25 26 27}]
```

由实验结果可以看出，mat3确实是多通道矩阵mat1的转置矩阵，且mat1的值没有被改变，实验完成。

计时的方式分析

由于本程序是在Linux环境下运行，所以使用Linux环境下的计时函数gettimeofday()进行程序的计时，该计时函数精度较高，可以达到微秒级别。
通过查看计时函数的结构可以发现，该结构体的定义为

```
struct timeval
{
  __time_t tv_sec;       /* Seconds.  */
  __suseconds_t tv_usec;    /* Microseconds.  */
};
```

这个函数获取从1970年1月1日到现在经过的时间和时区（UTC时间），但是按照Linux的官方文档，该时区已经不再使用，所以在使用的时候传入NULL即可。

在需要计时的函数开头以及结尾分别获取一次当前时间，它们之间的差值即为程序运行的时间。
操作代码如下

```
    gettimeofday(&read_t11, NULL);
   //function
    gettimeofday(&read_t12, NULL);
    read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)
(read_t12.tv_usec - read_t11.tv_usec) / 1000000.0;
```

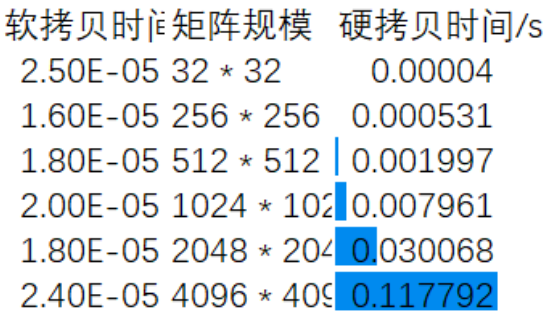
Hard Copy 与 Soft Copy 时间差异比较

通过查阅资料发现硬拷贝会比软拷贝在时间上效率低很多，因为软拷贝在创建新的矩阵对象的时候并不会赋值大量的空间，但不知道具体会在效率上面快多少，于是进行下面一组实验，以返回子矩阵为例，探究硬拷贝和一般的ROI操作在创建矩阵效率上面的区别。在本实验中，分别以不同规模的数组为实验对象，分别返回他们的同等大小的子矩阵，比较创建同等大小子矩阵所需要的时间。

实验结果如下表格所示：

软拷贝时间/s	矩阵规模	硬拷贝时间/s
2.50E-05	32 * 32	0.00004
1.60E-05	256 * 256	0.000531
1.80E-05	512 * 512	0.001997
2.00E-05	1024 * 1024	0.007961
1.80E-05	2048 * 2048	0.030068
2.40E-05	4096 * 4096	0.117792

为了更加直观地观察出硬拷贝与软拷贝在效率上的区别，将该表格绘制成蝴蝶图，如下图所示：



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10^5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。

多通道矩阵乘法的访存优化探索

对于多通道的矩阵乘法，与一般的矩阵乘法相比多了通道数这一性质，所以在存储的时候为了便于同一个元素不同通道的访问，将同一个元素不同通道的值都存储在相邻的位置上，而在朴素的矩阵乘法当中，使用的是分别对每一通道的元素进行单独运算。朴素代码如下所示：

```
for(int ch = 1; ch <= channel; ch ++){//不同通道数的切换
          for(int i = 0; i < row; i ++){
             for(int j = 0; j < mat.col; j ++){
                 for(int k = 0; k < mat.row; k ++){
                     res(i, j, ch) += (*this)(i, k, ch) * mat(k, j,
ch);
                 }
             }
         }
     }
```

通过对于不同规模的矩阵进行运算测试，发现朴素实现的多通道乘法的速度过慢导致2048 * 2048的矩阵运算时间都已经超过了七分钟。

现进行访存优化多通道矩阵乘法的探索

由于数据都是2的倍数，所以以双通道为例进行试验的探究。

进一步考虑，由于在矩阵类当中，每一元素不同通道的值都是存储在一块相邻的区域当中，所以考虑将最外层的通道切换移动到循环的最内层继续进行试验。

通过将相邻的通道数放在循环的最内侧，将外侧的三层循环按照i, k, j的方式组合，使得在最内层的计算每次都是按照顺序地进行访问，进一步实验。

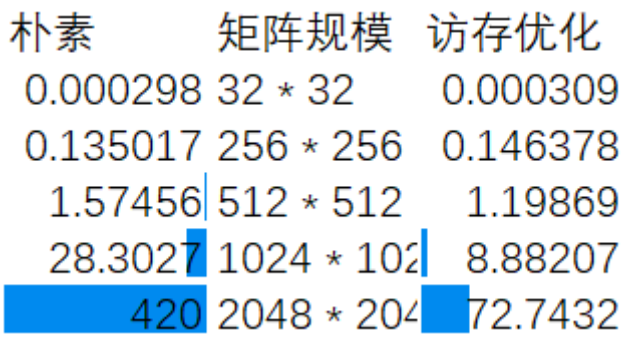
三次的实验结果如下表格所示：

矩阵规模	32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048
ch_i_j_k时间	0.000298	0.135017	1.57456	28.3027	...
i_j_k_ch					

时间	0.000385	0.142711	1.47202	16.6255	...
矩阵规模	32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048

i_k_j_ch 时间	0.000309	0.146378	1.19869	8.88207	72.7432

通过制作朴素多通道矩阵乘法与访存优化多通道矩阵乘法的蝴蝶图进行更直观地比较（如下图所示）



和前两次项目相比，这次的对比试验让我对访存优化的理解更为深刻，仅仅是改变了计算机访问元素的顺序即可在硬件级别上使得多通道的矩阵乘法效率更高。由于数组是一段连续的内存，在本程序当中由于使用一维数组来表示二维数组加上不同通道的值，通过模拟二维数组当中的行之间的跳转访问操作来实现不同元素不同通道的跳转。将操作最频繁的切换通道操作放在循环的最内层，并且改变对两个矩阵行列的访问次序可以显著地优化计算机访问元素的次序从而加速多通道矩阵的乘法。

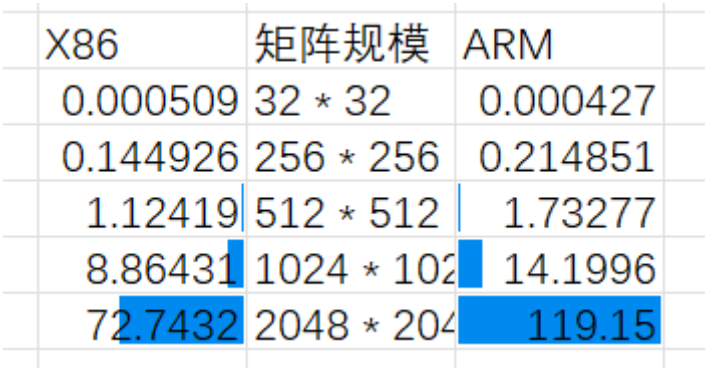
代码在ARM服务器运行与在X86架构计算机上运行的差异比较

单通道矩阵乘法的比较（计算能力的比较分析）

通过访存优化进行在两种平台上的比较分析，实验结果如下表所示：

使用较好访存优化					
矩阵规模	32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048
X86	0.000509	0.144926	1.12419	8.86431	72.7432
ARM	0.000427	0.214851	1.73277	14.1996	119.15

为了更直观地展现结果，将该表绘制成为蝴蝶图如图所示：



可以发现随着矩阵规模的增大在ARM上运行的时间始终要比X86下运行的时间要多，且在矩阵规模为2048*2048时，运行时间大概为X86的1.65倍，且随着矩阵规模的增大，该比例还会上升。

获取子矩阵的硬拷贝比较（内存分配以及访问性能比较）

通过实现获取子矩阵的硬拷贝方法，比较两种架构下计算机的内存分配性能，实验结果如下所示：

获取子矩阵硬拷贝时间			

获取子矩阵硬拷贝时间			
矩阵规模	32 * 32	256 * 256	2048 * 2048
X86	4.50E-05	0.000721	0.03336
ARM	1.8e-05s	5.88E-04	0.04501s

可以发现，二者在内存分配方面的差异并不是非常明显，几乎持平。

综上分析，X86架构的计算机性能比ARM要好一些，经过查阅资料发现，原因是X86的CPU核数更多，制备工艺更好，计算资源更多导致，而ARM的优势在于功耗低，专一性强，二者各有优势。

Part4 Difficuties and Solutions

在使用重载二目运算符时对主调对象的值进行误修改

在一次进行矩阵加法的实验中，程序出现了逻辑错误，经过查看，发现原因是错误地在函数中进行了对原矩阵值的修改，所以应当在不希望修改原对象的值的函数头使用`const`限定符，以此来保证程序运行的安全可靠。

在程序当中显式调用对象的析构函数但是无法释放对象

```
int main(){
    int *nums1 = new int[30];
    for(int i = 0; i < 30; i++){
        nums1[i] = i + 1;
    }
    Matrix<int> mat1(5, 6, 1, nums1);
    Matrix<int> mat2(2, 5, 3, nums1);
    Matrix<int> mat3 = mat1;
    mat1.~Matrix();

}
```

在进行第三部分的实验当中，为了检验内存管理的安全性，在程序当中试着显式地调用矩阵对象mat1的析构函数，实验结果片段如下：

```
Destructor has been invoke.
-1 member(s) exist(s)
-1 matrices share the element array.
```

由上述结果可以得知，即使在程序内部显式调用了析构函数，在对象声生命周期结束之后，程序仍然还会再次调用析构函数，释放该对象一次，这样就会导致现有对象的数目变为-1，经过查阅资料(C++ Primer Plus)，发现在程序内部，编译器将我们自己定义的析构函数当作了普通成员函数处理，实际上不像构造函数，我们自己定义析构函数的时候，不会覆盖系统提供的析构函数，而系统提供的析构函数是负责对象的清理的。编译器允许显式调用析构函数，但是这种调用仅仅是调用了我们定义的析构函数（也就是当作普通成员函数处理的那个版本），并没有调用系统的析构函数。所以只是对象内部的动态分配的空间得到了释放，对象本身并不会因为显式调用析构函数而被销毁。所以可以使用代码块来显式地调用系统的析构函数来进行问题的解决。

Part5 Summarizing and Thinking

在本次项目当中，我觉得又比前一个项目更有了经验以及收获，首先是更加懂得了一个类的应该如何更完美地设计，保证内存不泄露的同时还要使得类的运行效率更高。也通过参考OpenCV文档更加了解了一个矩阵类应当如何设计。由于在现实生活当中一个矩阵类需要支持多种数据类型的矩阵，所以通过模板类的使用，可以达到这一目的，在这一过程当中，我也不懂模板类到熟练操纵模板类完成自己想要做的事情，这一点的进步也是非常大的。从软拷贝也硬拷贝的比较当中，我也更加直观地感受到了两者效率上的区别，在ARM服务器上运行自己的代码也让我对服务器更加了解。

以上就是我的报告全部内容

感谢这次项目!感谢老师的阅读