CS205 C/ C++ Programming - Lab Assignment Template

Name:舒子和
SID:12111727

Part 1 – Analysis

一、文件说明:

1. test.cpp
2. matrix.hpp

其中 test.cpp 是对 matrix 库进行测试的文件，其中有 main 函数。

二、需求分析:

a) 设计一个 matrix 类，该矩阵含有行数列数和通道数来存储数字信息，且数字类型可为很多种。

解决思路：利用类模板 `template<class T,size_t channel>`
`class matrix`
参照 opencv，模板类的类型尖括号内代表矩阵的存储数字类型和通道数。

b) 当对类进行赋值或拷贝时用软拷贝来加快程序运行效率减少内存使用，同时需要注意内存申请释放。参照 opencv，考虑到软拷贝的矩阵共用一个信息，当修改一个矩阵的信息时另一个也会改变，故也引入 matrix::clone 函数硬拷贝。

c) 进行各种运算符的重载，相较于 Project3、Project4，本次项目拥有多个通道，计算更加复杂。
同时重载+，-，*法完成常数和矩阵的加减乘。

d) 参考 Opencv，引入 ROI 截取子矩阵，并且操作指针避免硬拷贝。

其中矩阵乘法的原理参考 Project4，对矩阵数字内存尽量地连续访问，加快程序的运行速率。

Part 2– Code

```
template<class T,size_t channel>
class matrix
{
private:
    size_t rows,cols;
    T *data;
```

```
    size_t *refcount;
    size_t channels=channel;
    size_t step;
public:
    bool isvalid=false;
}
```

矩阵的类模板，支持各种类型数据的存储，类型里的参数 channel 代表定义矩阵的通道数，rows,cols 代表矩阵的行与列数，指针 data 代表矩阵的数据，指针 refcount 内存申请于 data 之前（连续的），用于解决软拷贝的内存管理问题。step 代表矩阵需要换行需要读多少步，用于 ROI 矩阵数据的读取运算。Isvalid 用于标记矩阵是否具有合法性，在构造和运算时会检测与修改矩阵的合法性。

```
matrix(size_t rows,size_t cols,const T
*data):rows(rows),cols(cols),step(cols*channel)
    {
        if(rows>0&&cols>0&&this->channels>0){
        this->refcount=(size_t *)(new
char[rows*cols*channels*sizeof(T)+sizeof(size_t)]{});
        this->data=(T *)((char *)this->refcount+sizeof(size_t));
        for(size_t i=0;i<rows*cols*channels;i++)
        {
            this->data[i]=data[i];
        }
        this->refcount[0]=1;
        this->isvalid=true;
        }
//...
```

矩阵的构造函数，申请长度为 rows*cols*channel*sizeof(T)+4 的内存长度，data 指向前者，refcount 指向头部的 4 位（size_t），并且初始化为 1，在后续的赋值，拷贝和析构中会改变 refcount[0]的值；

```
    ~matrix(){
        //...

        if(this->refcount[0]<=1){
        delete[] this->refcount;
        }
        else{
        this->refcount[0]--;
        }
    }
```

矩阵的析构函数，当引用同一处 data 的矩阵数目多于 1 个时，并不执行内存的释放，而是使 refcount[0]-1，直到析构的矩阵只有最后一个时，才执行 delete[]进行内存释放。

```
template<class T,size_t channel>
matrix<T,channel>::matrix(const matrix& mat)
{
    if((!mat.data)||mat.isvalid==false){
        cout<<"null data"<<endl;
        this->isvalid=false;
    }
    else{
    this->cols=mat.cols;
    this->rows=mat.rows;
    this->channels=mat.channels;
    this->refcount=mat.refcount;
    this->data=mat.data;
    this->refcount[0]++;
    this->step=mat.step;
    this->isvalid=true;
    }
//...

}
```

拷贝操作，进行矩阵合法性判断后直接给指针赋值，然后将 refcount[0]++，赋值操作同理，不同之处在于赋值操作在赋值前需要将原矩阵所存的内存空间进行释放，以防止内存泄漏。

```
    matrix operator*(const matrix& mat);

    template<class CT,size_t channel_,typename Q>
    friend matrix<CT,channel_> operator*(Q num,const
    matrix<CT,channel_>& mat);

    template<class CT,size_t channel_,typename Q>
    friend matrix<CT,channel_> operator*(const matrix<CT,channel_>&
mat,Q num);
```

对乘法运算符进行重载，需要注意的是对于与矩阵存储数据类型不同的常数进行乘法操作时，此处在类模板中使用了函数模板（故有 T,Q 两个 typename），从而实现对常数类型的转化然后计算。加法减法同理。

此处展示乘法：

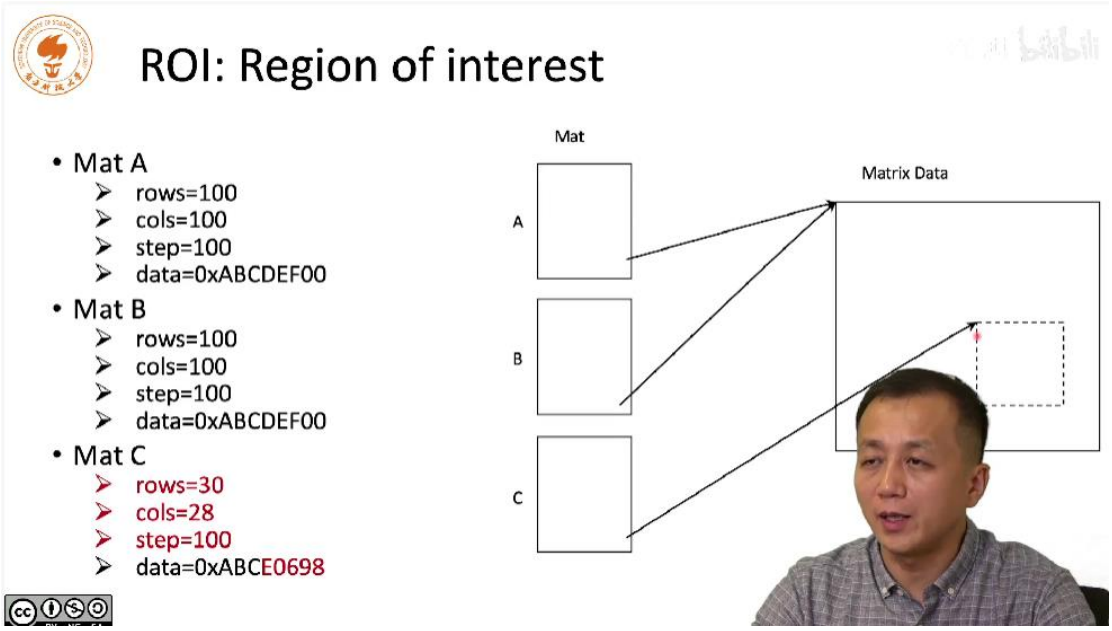
```
template<class T,size_t channel>
matrix<T,channel> matrix<T,channel>::operator*(const matrix<T,channel>&
mat)
{
    if(this->cols==mat.rows&&this->channels==mat.channels&&this->data!=0
&&mat.data!=0&&this->isvalid&&mat.isvalid)
    {
        T temp;
        T data[this->rows*mat.cols*channel]={(T)0};

        for(int i=0;i<this->rows;i++){
            for(int t=0;t<channel;t++){
            for(int k=t;k<this->cols*channel;k+=channel){
                temp=this->data[i*this->step+k];
                for(int j=t;j<mat.cols*channel;j+=channel){
                    data[i*this->cols*this->channels+j]+=temp*mat.data[(
k/channel)*mat.step+j];
                }
            }
        }
        }
        matrix<T,channel> matr(this->rows,mat.cols,data);
        matr.isvalid=true;
        return matr;
    }
    else
    {
        cout<<"operation * invalid"<<endl;
        return *this;
    }
}
```

乘法操作的难点在于，与之前不同，本次作业有多个通道，为了保证程序运行的效率，仍使用一维指针进行存储，且在后续有 ROI 子矩阵进入运算时，逻辑更复杂。
矩阵的逻辑算法引入了 channel（通道数），和 step（子矩阵的换行），更加复杂。

```
matrix(const matrix& mat,const Rectangle& rect);

struct Rectangle{
    size_t row,col,width,height;
};

Rectangle Rect(size_t row,size_t col,size_t width,size_t height)
```

```
{
    Rectangle ref;
    ref.row=row;
    ref.col=col;
    ref.width=width;
    ref.height=height;
    return ref;
}

template<class T,size_t channel>
matrix<T,channel>::matrix(const matrix& mat,const Rectangle& rect)
{
    if((!mat.data)||mat.isvalid==false){
        cout<<"null data"<<endl;
        this->isvalid=false;
    }
    else
if(rect.row+rect.height>mat.rows||rect.col*+rect.width>mat.cols)
    {
        cout<<"submatrix is out of range"<<endl;
        this->isvalid=false;
    }
    else{
    this->rows=rect.height;
    this->cols=rect.width;
    this->channels=channel;
    this->refcount=mat.refcount;
    this->data=mat.data+rect.col*channel+rect.row*mat.step;
    this->step=mat.step;
    this->refcount[0]++;
    cout<<"ROI is used"<<endl;
    this->isvalid=true;
    }
}
```

参考 Opencv，将 ROI 取子矩阵的操作写为构造函数，同时定义一个存储了一个矩形信息的结构体，用于框取子矩阵。在判断取 ROI 的操作合法时，将子矩阵的 data 指针指向大矩阵申请内存的中间，从而避免硬拷贝，同时 refcount[0]也要加一，因为该子矩阵实际也共用同样的内存中的一部分。




```
template<class T,size_t channel>
class matrix
{
private:
    size_t rows,cols;
    T *data;
    size_t *refcount;
    size_t channels=channel;
    size_t step;
public:

    bool isvalid=false;

    matrix(size_t rows,size_t cols,const T
*data):rows(rows),cols(cols),step(cols*channel)
    {
        if(rows>0&&cols>0&&this->channels>0){
        this->refcount=(size_t *)(new
char[rows*cols*channels*sizeof(T)+sizeof(size_t)]{});
        this->data=(T *)((char *)this->refcount+sizeof(size_t));
        for(size_t i=0;i<rows*cols*channels;i++)
        {
            this->data[i]=data[i];
        }
        this->refcount[0]=1;
        this->isvalid=true;
```

```
        }
        //...
        else
        {
            this->isvalid=false;
            this->data=0;
            cout<<"invalid matrix."<<endl;
        }
    }

    matrix(const matrix& mat);

    matrix(const matrix& mat,const Rectangle& rect);

    size_t get_channels(){return channels;}

    void print_matrix();

    bool operator=(const matrix& mat);

    matrix operator+(const matrix& mat);

    template<class CT,size_t channel_,typename Q>
    friend matrix<CT,channel_> operator+(Q num,const
matrix<CT,channel_>& mat);

    template<class CT,size_t channel_,typename Q>
    friend matrix<CT,channel_> operator+(const matrix<CT,channel_>&
mat,Q num);

    bool operator==(const matrix& mat);

    matrix operator-(const matrix& mat);

    template<typename Q>
    matrix operator-(Q num);

    matrix operator*(const matrix& mat);

    template<class CT,size_t channel_,typename Q>
    friend matrix<CT,channel_> operator*(Q num,const
matrix<CT,channel_>& mat);

    template<class CT,size_t channel_,typename Q>
```

```
    friend matrix<CT,channel_> operator*(const matrix<CT,channel_>&
mat,Q num);

    bool clone(const matrix& mat);

    ~matrix(){
        cout<<"des~ has been run :"<<this->data[0]<<endl;
        if(this->refcount[0]<=1){
        delete[] this->refcount;
        cout<<"data has been deleted."<<endl;
        }
        else{
        this->refcount[0]--;
        cout<<"data is still alive :"<<this->refcount[0]<<endl;
        }
    }
};
```

一整个类模板。

Part 3 - Result & Verification

```
    int data[27]={255,0,0,255,0,0,255,0,0,0,255,0,0,255,0,0,255,0
    ,0,0,255,0,0,255,0,0,255};
    matrix<int,3> mat1(3,3,data);
    mat1.print_matrix();
```

OUT:

```
] [255 0 0 ] [255 0 0 ] [255 0 0
] [0 255 0 ] [0 255 0 ] [0 255 0
] [0 0 255 ] [0 0 255 ] [0 0 255
```

```
    int data[27]={255,0,0,255,0,0,255,0,0,0,255,0,0,255,0,0,255,0
    ,0,0,255,0,0,255,0,0,255};
    int data2[27]={};
    matrix<int,3> mat1(3,3,data);
    matrix<int,3> mat2(3,3,data2);
    mat2=mat1;
    mat2.print_matrix();
```

OUT:

```
= is used
] [255 0 0 ] [255 0 0 ] [255 0 0
] [0 255 0 ] [0 255 0 ] [0 255 0
] [0 0 255 ] [0 0 255 ] [0 0 255

des~ has been run :255
data is still alive :1
des~ has been run :255
data has been deleted.
```

```
cout<<(mat1==mat2)<<endl;
```

OUT:

```
0
```

```
(mat1+mat2).print_matrix();
```

OUT:

```
] [256 1 1 ] [255 0 0 ] [255 0 0
] [0 255 0 ] [1 256 1 ] [0 255 0
] [0 0 255 ] [0 0 255 ] [1 1 256
```

```
(mat1-mat2).print_matrix();
```

OUT:

```
] [254 -1 -1 ] [255 0 0 ] [255 0 0
] [0 255 0 ] [-1 254 -1 ] [0 255 0
] [0 0 255 ] [0 0 255 ] [-1 -1 254
```

```
(mat1*mat2).print_matrix();
```

OUT:

```
] [255 0 0 ] [255 0 0 ] [255 0 0
] [0 255 0 ] [0 255 0 ] [0 255 0
] [0 0 255 ] [0 0 255 ] [0 0 255
```

```
matrix<int,3> mat3(mat1,Rect(1,1,2,2));
mat3.print_matrix();
```

OUT:

```
ROI is used
] [0 255 0 ] [0 255 0
] [0 0 255 ] [0 0 255
```

```
    mat1.clone(mat3);
    cout<<(mat1==mat3)<<endl;
```

OUT:

```
1
```

Part 4 - Difficulties & Solutions

1、 在给矩阵申请内存时的内存操作，软拷贝：

先申请一字节的内存单位（new char[]）总共申请 rows*cols*channels*sizeof(T)+sizeof(size_t) 大小，再将该指针强转为 size_t *指针，赋值给 refcount 指针，于是 refcount[0]就可以用前四位内存记录一个矩阵的个数。再将 refcount 指针+1 后强转为 T*指针，指向 data 存储的位置。

2、 重载运算符时，遇到矩阵和一个常数相加、减或乘时，这个常数的类型不一定要和矩阵存储的数字类型一样：

在类模板中写友元函数模板，

```
template<class CT,size_t channel_,typename Q>
friend matrix<CT,channel_> operator+(const matrix<CT,channel_>&
mat,Q num);
```

拥有两个 typename，不过发现在类模板中定义的 class T 在友元函数得换一个名称(class CT)。

3、 ROI 矩阵下的数据读取和计算：

由于 ROI 矩阵的指针指向大矩阵中的一处，在进行 ROI 矩阵的读数时，指针不会自动（换行），而是仍旧会读到大矩阵的数据，于是引入 step 成员，step 等于大矩阵的列数乘通道数，ROI 子矩阵与大矩阵共享一个 step，于是在读满 ROI 矩阵的列数时，手动利用 step*row 进行换行，实现 ROI 矩阵的数据读取和计算。