- $LaTeX$ of README.md may fail to display on GitHub. For better experience, pls check report in pdf format.

CS205 C/ C++ Programming Project05

A Class for Matrices

Name: 樊斯特(Fan Site)

SID: 12111624

项目结构

```
E:\CPP\PROJECT05
|   CMakeLists.txt
|   README.md
|
├─build
|       //makefile here
|
├─doc
|       Report.pdf
|
└─src
         benchmark.cpp
         data.hpp
         matrix.hpp
```

*编译时添加选项 -DSAFE 可以调用严格版函数实现。

Part 1 - Analysis

题目重述&主要思路

本题目要求使用C++语言实现一个具有一定功能的矩阵类。

根据题目描述，题目要求的矩阵乘法需要支持的主要功能为：

1. 支持多通道存储数据
2. 支持多种数据类型
3. 赋值时避免深拷贝(Hard Copy)，安全管理内存
4. 重载基本运算符
5. 不使用深拷贝实现ROI

本项目除完成上述基础要求外，支持以下内容：

6. **支持任意合理重载运算符的数据类型**
7. **跨数据类型运算、转换、赋值**(提供转换函数接口)
8. **异常处理机制**(另提供严格版函数)
9. **子矩阵/掩膜提取两类ROI实现**
10. **逐元素一元/二元自定义运算**(提供运算函数接口)

11. **默认重载软拷贝(Shallow Copy)，提供硬拷贝函数**
12. **更多易用重载运算符**
13. **用户友好的足量注释**

模型假设

项目要求矩阵类需要适用于不同数据类型，因此本项目主体使用类模板完成，笔者考虑该库所可能使用的数据范围作为本项目支持的数据规模，如下：

- 可适用的矩阵尺寸因元素类型而异，因此未设置`rows`和`cols`上限(不抛出`bad alloc exception`即可)。
- 通道数上限`MAX_CHANNEL`通过宏默认设置为4(适用于ARGB存储图像)，有更大需求修改宏即可
- 矩阵元素的数据兼容基本数据类型，适用于满足前置条件的自定义类型：
  - 无参数构造器，重载`=`，`==(!=)`
  - 与其他类型的隐式转换/显式的转换函数
  - 与不同类型进行运算的结果类型推导
- 多通道矩阵连续访问同一通道不同元素次数>同一元素不同通道值次数，因此选择对通道进行稀疏存储，即同一通道内行优先存储。

Part 2 - Code

本项目实现了严格版和普通版两版实现，安全性均可保证，但严格版的异常抛出信息更为具体，便于调试，此处展示严格版实现。

矩阵数据类

```
//data.hpp
//data class
template<typename Tp>
class data
{
private:
    Tp *value;
    size_t length;
    size_t *ref_count;
public:
    ...
};
```

本项目根据尽可能避免Hard Copy的要求，参考课上所述的形式将矩阵内数据以私有成员的形式安全封装为类模板，在以`Tp *`指针`value`存储数据头外，附加存储了`size_t`类型的数据长度`length`，以及`size_t *`类型的"数据复用次数"统计变量`ref_count`，表示该数据可被多少个对象调用，便于析构矩阵时安全地释放内存。

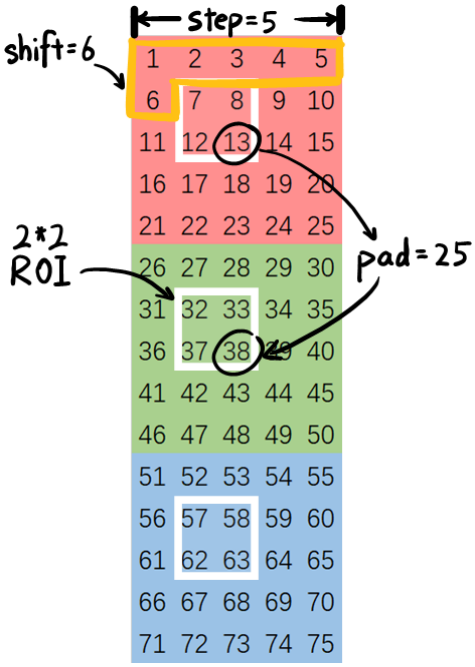
矩阵类

```
//matrix.hpp
//matrix class
#define MAX_CHANNEL (4)
#define MAX_CHANNEL (4)
```

```
template<typename Tp>
class matrix
{
private:
    typedef matrix<bool> mask;
    typedef char channel_number;

    size_t rows;
    size_t cols;
    //number of channels should be [1,MAX_CHANNEL]
    channel_number channels;
    //padding of relative position
    size_t channel_pad;
    size_t step;
    size_t shift;
    data<Tp> *dat;
public:
    ...
};
```

本项目的矩阵类实现为类模板，以私有成员的形式存储了包括行数、列数、通道数、内部数据的基本信息，其中行列数使用`size_t`存储，通道数限定在`[1, MAX_CHANNEL]`间，使用`char(channel_number)`存储，最大通道数默认为4，数据使用`data`类以行优先、通道稀疏存储，下图是一个`5*5`的RGB三通道矩阵在类中的存储方式：



根据题目对`Non-Hard-Copy ROI`的要求，为了复用同一`dat`，矩阵类多存了三个`size_t`类型变量：`shift`，`step`，`channel_pad`。

- `shift`：存储当前矩阵相对`dat->value`的数据起始位置
- `step`：存储该数据原本的列数
- `channel_pad`：跳转到下一通道该元素所需加的数，即单个通道的尺寸

构造函数

data

```
//data.hpp
template<typename Tp>
class data
{
    ...
public:
    //! constructor using the given source data
    explicit data(Tp *dat, size_t length);
    //! constructor that malloc new memory of the given length
    explicit data(size_t length);
    //! soft copy constructor
    data(const data &p);
    ...
}
```

对于`data`类，项目支持源数据指针+数据长和仅传入数据长两种构造器以及软拷贝构造器(直接传递`value`指针)，此处展示**仅传入数据长的构造器。**

```
//! constructor that malloc new memory of the given length
template<typename Tp>
data<Tp>::data(size_t length):length(length)
{
    try
    {
        value = new Tp[length];
        ref_count = new int[1]{1};
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << length << "\n";
        throw std::bad_alloc();
    }
}
```

仅传入数据长的构造器要求`Tp`类型需要有无参构造器，在为`value`申请空间时能捕获异常，通过错误流输出报错信息。

matrix

```
//matrix.hpp
template<typename Tp>
class matrix
{
    ...
public:
    //! constructor that sets matrix elements to given source data
    matrix(size_t rows, size_t cols, Tp *src, channel_number channels = 1);
    //! constructor that sets each matrix element to specified value
```

```
    matrix(size_t rows, size_t cols, const Tp &value, channel_number channels =
1);
    //! submatrix constructor(ROI)
    matrix(matrix &src, size_t row1, size_t col1, size_t row2, size_t col2);
    //! copy constructor (soft copy)
    matrix(const matrix &p);
    ...
}
```

`matrix`严格意义上的构造器有4个，分别是尺寸+数据源、尺寸+单个值、子矩阵、软拷贝构造器，此处展示**尺寸+数据源构造器**和**软拷贝构造器**，子矩阵构造器会在ROI部分详讲。

```
//! constructor that sets matrix elements to given source data
template<typename Tp>
matrix<Tp>::matrix(size_t rows, size_t cols, Tp *src, channel_number
channels):rows(rows), cols(cols), channels(channels)
{
    if (rows * cols == 0)
    {
        throw std::invalid_argument("Invalid Argument Exception: row number and
column number should be positive.\n");
    }
    if (channels > MAX_CHANNEL || channels <= 0)
    {
        throw std::invalid_argument("Invalid Argument Exception: channel number
should be in [1,MAX_CHANNEL].\n");
    }
    this->shift = 0;
    this->step = cols;
    this->channel_pad = rows * cols;
    this->dat = new data<Tp>(src, size());
}
```

在生成与其他矩阵不存在包含关系的新矩阵时，将`shift`置零，`step`设为`cols`，`channel_pad`设为`rows*cols`，使用数据源创建`data`即可，过程中对矩阵的尺寸也做了检查和异常处理。

```
//! copy constructor (soft copy)
template<typename Tp>
matrix<Tp>::matrix(const matrix &p)
{
    if (p.get_dat() == nullptr || p.get_dat() == NULL)
    {
        throw std::invalid_argument("Null Pointer Exception: The data of source
matrix is null.\n");
    }
    rows = p.get_rows();
    cols = p.get_cols();
    channels = p.get_channels();
    step = p.get_step();
    shift = p.get_shift();
    dat = p.get_dat();
    dat->inc_ref_count();
}
```

软拷贝构造器，直接传递成员即可，同时将被引用的数据的`ref_count`自增。使用getter是因为原计划再包一个template，支持不同模板类之间的拷贝(不同模板类之间私有成员互不可见)，但转换过程是硬拷贝的，因此未在此函数实现跨类复制。

内存管理

```
//! increase refcount
template<typename Tp>
void data<Tp>::inc_ref_count()
{
    (*ref_count)++;
}

//! decrease refcount
template<typename Tp>
void data<Tp>::dec_ref_count()
{
    (*ref_count)--;
    if (!(*ref_count))
    {
        this->~data();
    }
}
```

为避免硬拷贝，同一块数据可能同时被若干个矩阵中的`data *`指向。在创建/复制矩阵时，若使用了已经存在的data，则调用`inc_ref_count`，在删除矩阵时，不能直接释放数据，而是调用`dec_ref_count`，待`ref_count`归零，即这块数据没有被任何地方调用时才能释放。

这里的`ref_count`也使用指针的原因是当data被引用时，加入一个新的引用对象调用的`inc_ref_count`需要作用于所有指向`value`这块内存的data上，即便数据头有所不同。

类型转换函数

```
template<typename T1, typename T2>
T1 default_convert(T2 src)
{
    return (T1) src;
}

//customized type conversion
template<typename Tp>
template<typename target_type>
matrix<target_type> matrix<Tp>::convert_to(target_type (*convert_function)(Tp
src)) const
{
    try
    {
        target_type *NewArr = new target_type[size()]{};
        size_t it = 0;
        for (matrix::channel_number ch = 1; ch <= channels; ch++)
        {
            for (size_t r = 1; r <= rows; r++)
            {
                for (size_t c = 1; c <= cols; c++)
```

```
                {
                    NewArr[it++] = convert_function(at(r, c, ch));
                }
            }
        }
        return matrix<target_type>(rows, cols, NewArr, channels);
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << size() << "\n";
        throw std::bad_alloc();
    }
}
```

此处展示类型转换函数的实现，此函数可根据传入的**转换函数指针**`convert_function`将`matrix<Tp>`转换为`matrix<target_type>`，过程由于产生了类型转换而无法避免硬拷贝。

在类模板声明中，函数参数的默认值为显式转换函数`default_convert`，因此只要使用支持显式转换的基础类型(int, float...)，或是用户所给的类型间重载了显式转换，则可以不用传参数直接使用转换函数。

当然，如果用户希望使用自己的转换方式，例如手写`int round(float)`、`double ln(int)`等，也可以将自己的转换传入提供的参数接口即可：

```
int my_round(float x);
matrix<float> mat(4,4,src,3);
mat=mat.convert_to(myround);
//这里右值是matrix<int>，左值是matrix<float>，赋值过程隐式调用了convert_to<float>
(default_convert)
```

实际上用这个接口就能实现**任意逐元素一元运算操作**，上面提的两个例子就是逐元素四舍五入和逐元素取对数，一切交给用户的想象力！

重载赋值符

```
//! override same type assign operator(soft copy)
template<typename Tp>
matrix<Tp> &matrix<Tp>::operator=(const matrix<Tp> &p)
{
    if (this == &p)
    {
        return (*this);
    }
    if (p.dat == nullptr || p.dat == NULL)
    {
        throw std::invalid_argument("Null Pointer Exception: The data of source
matrix is null.\n");
    }
    rows = p.rows;
    cols = p.cols;
    channels = p.channels;
    step = p.step;
    shift = p.shift;
    channel_pad = p.channel_pad;
```

```
    if (dat != p.dat)
    {
        dat->dec_ref_count();
    }
    dat = p.dat;
    dat->inc_ref_count();
}
```

首先是同类型的赋值，彼此可以访问私有成员，在检查右值合法性后，将左值数据的引用数自减，将右值逐成员复制即可。

```
//! override implicit type transform assign operator(convert & hard copy)
template<typename Tp>
template<typename rhs>
matrix<Tp> &matrix<Tp>::operator=(const matrix<rhs> &p)
{
    if (p.get_dat() == nullptr || p.get_dat() == NULL)
    {
        throw std::invalid_argument("Null Pointer Exception: The data of source
matrix is null.\n");
    }
    dat->dec_ref_count();
    (*this) = p.template convert_to<Tp>();//调用默认转换器
    return (*this);
}
```

其次是跨类型的赋值，彼此的私有成员不可见，因此使用getter，在检查右值合法性后，将左值数据的引用数自减，将右值逐成员复制即可。

由于等式左右类型不同，函数会调用默认转换器，如果用户此时定义好了显式类型转换，赋值时就会**隐式调用显式转换**来达成赋值的目的。左右类型一定是不同的，因此左值的数据需要释放，引用数自减，根据转换后的右值新建一块数据。

ROI

本部分在参观`OpenCV::mat`后实现，借鉴了课程所述的方式实现了`ROI`。

由于时间和技术力有限，笔者在参观`OpenCV::mat`的多边形、圆形ROI后并未能成功复现仅传入`vector<point>`即可根据凸包生成`ROI`的高超函数，但此处实现了基础的**矩形ROI**和**掩膜ROI**两种形式，后者的可拓展性很强，只要配合一个能根据`vector<point>`输出某个元素是否在凸包内的函数即可实现任意形状的ROI的效果。

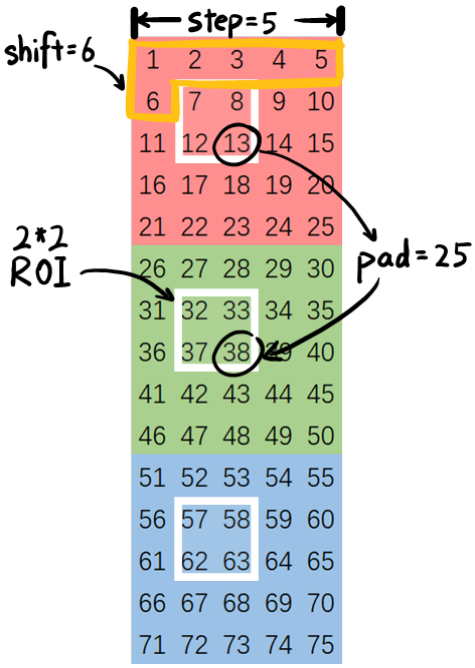
矩形ROI

```
//! submatrix constructor(ROI)
template<typename Tp>
matrix<Tp>::matrix(matrix &src, size_t row1, size_t col1, size_t row2, size_t
col2)
{
    if (row1 > row2 || col1 > col2)
    {
        throw std::out_of_range("Out Of Range Exception: row2 and col2 should be
greater than row1 and col1.\n");
    }
```

```
    if (row2 > src.rows || col2 > src.cols)
    {
        throw std::out_of_range("Out Of Range Exception: row2 and col2 exceed
the size of source matrix.\n");
    }
    if (row1 * col1 == 0)
    {
        throw std::out_of_range("Invalid Argument Exception: row number and
column number should be positive.\n");
    }
    if (src.get_dat() == nullptr || src.get_dat() == NULL)
    {
        throw std::invalid_argument("Null Pointer Exception: The data of source
matrix is null.\n");
    }
    rows = row2 - row1 + 1;
    cols = col2 - col1 + 1;
    channels = src.channels;
    step = src.step;
    channel_pad = src.channel_pad;
    shift = (row1 - 1) * step + col1 - 1;
    dat = src.get_dat;
    dat->inc_ref_count();
}
```

实现原理说起来也很简单，构造时推导出`shift`，`step`以及`channel_pad`并存储，在取出矩阵元素时再加上行数、列数、通道数总共六个参数即可算出`mat[i][j][channel]`在一维数组`value[]`中的位置，此处用**重载括号**的方式实现：

```
//! get element reference from matrix
template<typename Tp>
Tp &matrix<Tp>::operator()(size_t row, size_t col, channel_number channel) const
{
    if (row > rows || col > cols || channel > channels)
    {
        throw std::out_of_range(
                "Out Of Range Exception: arguments should be in the range of
source matrix.\n");
    }
    return (*dat)[shift + (channel - 1) * channel_pad + (row - 1) * step + col -
1];
}
```



上图所示的例子是从`5*5`三通道矩阵中取出一个`2*2`三通道ROI，图中的38对应的是ROI的**第二通道第二行第二列**的元素，经计算`roi(2,2,2)==dat[37]=38`，取出即可。

重载括号的传回值为引用类型，用户可以直接访问和修改矩阵元素。为了防止用户犯傻和便于部分声明为const的函数，项目还提供了只读版：

```
//! get element from matrix (read only)
template<typename Tp>
Tp matrix<Tp>::at(size_t row, size_t col, channel_number channel) const
{
    if (row > rows || col > cols || channel > channels)
    {
        throw std::out_of_range(
                "Out Of Range Exception: arguments should be in the range of
source matrix.\n");
    }
    return (*dat)[shift + (channel - 1) * channel_pad + (row - 1) * step + col -
1];
}
```

掩膜ROI

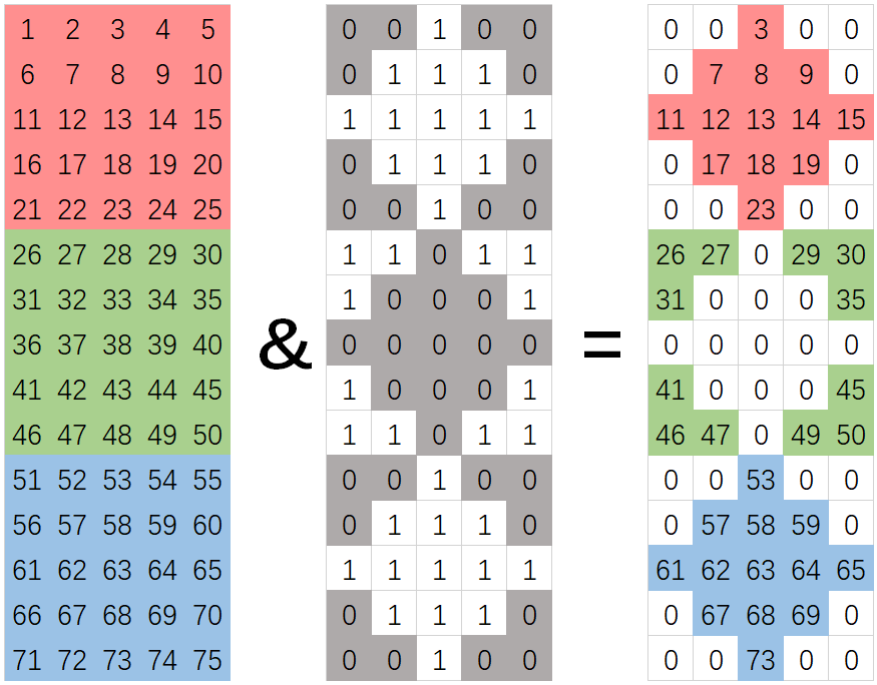
不知道读者有没有注意到matrix类中的`typedef matrix<bool> mask`，重命名了零一矩阵，实际上`OpenCV::mat`中也有类似的实现，即将零一矩阵作为**掩膜**，其中为1的部分即为ROI。

这里笔者模仿OpenCV实现了`copy_to`函数，根据当前矩阵和掩膜填充目标矩阵。出于对原数据的负责态度，此处未使用软拷贝(因为这样会修改原数据导致引用该数据的所有矩阵内容变化，而我们只是希望取出一块我们感兴趣的数据进行操作而已)。

```
//! ROI using mask(matrix<bool>)
template<typename Tp>
bool matrix<Tp>::copy_to(matrix<Tp> &dst, const matrix::mask &mask)
{
    try
    {
        dst.~matrix();
```

```
        Tp *NewArr = new Tp[size()]{};
        size_t it = 0;
        for (channel_number ch = 1; ch <= channels; ch++)
        {
            for (size_t r = 1; r <= rows; r++)
            {
                for (size_t c = 1; c <= cols; c++)
                {
                    if (mask(r, c, ch))
                    {
                        NewArr[it] = at(r, c, ch);
                    }
                    it++;
                }
            }
        }
        dst = matrix<Tp>(rows, cols, NewArr, channels);
        return true;
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << size() << "\n";
        throw std::bad_alloc();
    }
}
```

所以只要知道如何根据需求生成零一矩阵，就能通过调用copy_to函数取出ROI了。



通道提取

掩膜也是多通道的，所以只想要某个通道的内容可以遮住其他通道，但是有时候我们真的只关心其中的某个通道，比如透明度，本项目为此实现了`Channel Of Interest`效果的函数：

```
//! split out a single channel
template<typename Tp>
```

```
matrix<Tp> matrix<Tp>::split_channel(matrix::channel_number channel_id)
{
    if (channel_id > channels || channel_id <= 0)
    {
        throw std::out_of_range(
                "Out Of Range Exception: channel id should be within the channel
number of source matrix.\n");
    }
    matrix<Tp> New(*this);
    New.channels = 1;
    New.shift += channel_pad * (channel_id - 1);
    return New;
}
```

在这里，我们通过将`shift`后推若干个`channel_pad`，让数据头起点到达目标通道，再将通道数置一，即剥离出了对应通道，然后我们就可以对多个通道间进行各种各样的运算和变换了。

考虑到剥离通道后通常需要进行不影响原数据的操作，因此此处选择使用硬拷贝取出数据。若希望对原数据进行操作就更简单了，直接对原矩阵的`shift`后推，`channels`置一即可，不过在操作后需要调整回原值。如果真的要为软拷贝做到这么复杂，为什么不用重载的括号，指定行列通道后直接进行操作呢?

重载==与比较

项目重载了`==`，用于严格地比较两个矩阵的各元素是否相同，实现非常简单：

```
//! override equation(strict equal)
template<typename Tp>
bool matrix<Tp>::operator==(const matrix &p) const
{
    if (this == &p)
    {
        return true;
    }
    if (rows != p.get_rows() || cols != p.get_cols() || channels !=
p.get_channels())
    {
        return false;
    }
    for (channel_number ch = 1; ch <= channels; ch++)
    {
        for (size_t r = 1; r <= rows; r++)
        {
            for (size_t c = 1; c <= cols; c++)
            {
                if (at(r, c, ch) != p.at(r, c, ch))
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```

但众所周知，数据类型的比较向来不那么简单。例如，对于基础数据类型`float`，直接使用`==`经常会因为误差而错判相等。但是用户又不想或是不能为此将一整个类型的`==`重载为更弱的比较条件，此时就要用到`equals`函数了：

```
//! equal with customized compare function
template<typename Tp>
bool matrix<Tp>::equals(const matrix<Tp> &p, bool (*equal)(Tp, Tp)) const
{
    if (this == &p)
    {
        return true;
    }
    if (rows != p.get_rows() || cols != p.get_cols() || channels !=
p.get_channels())
    {
        return false;
    }
    for (channel_number ch = 1; ch <= channels; ch++)
    {
        for (size_t r = 1; r <= rows; r++)
        {
            for (size_t c = 1; c <= cols; c++)
            {
                if (!equal(at(r, c, ch), p.at(r, c, ch)))
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```

本项目的`equals`函数支持用户自行传入函数，例如`float`的例子就可以：

```
bool float_eq(float x, float y)
{
    return (x-y<1e-3)&&(y-x<1e-3);
}
matrix<float> mat1(3,3,1.0001f);
matrix<float> mat2(3,3,1.0f);
bool eq=mat1.equals(mat2,float_eq);
```

可拓展性很强：哪怕用户传入的函数说0和1是相等的，`equals`也会忠诚地认为0和1是相等的！

重载加减乘

如果说实现稳健的运算是Project03的亮点，高效率是Project04的亮点，那本项目的亮点则在于：类模板让跨类型成为可能，因此本项目中重载的运算均支持不同类型矩阵的运算，此处仅展示朴素乘法。

```
template<typename T1>
template<typename T2>
matrix<decltype(T1() * T2())> matrix<T1>::operator*(matrix<T2> &p)
```

```
{
    if (cols != p.get_rows() )
    {
        throw std::invalid_argument(
                "Invalid Argument Exception: The col number of left matrix should
equal the row number of right matrix.\n");
    }
    if(channels != p.get_channels())
    {
        throw std::invalid_argument(
                "Invalid Argument Exception: The channel number of two matrix
should be the same.\n");
    }
    try
    {
        typedef decltype(T1() * T2()) result_type;
        auto *NewArr = new result_type[rows * p.get_cols() * channels]{};
        matrix<result_type> New(rows, p.get_cols(), NewArr, channels);
        for (size_t ch = 1; ch <= channels; ch++)
        {
            for (size_t i = 1; i <= rows; i++)
            {
                for (size_t k = 1; k <= cols; k++)
                {
                    for (size_t j = 1; j <= p.get_cols(); j++)
                    {
                        New(i, j, ch) = New.at(i, j, ch) + at(i, k, ch) *
p.at(k, j, ch);
                    }
                }
            }
        }
        return New;
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << size() << "\n";
        throw std::bad_alloc();
    }
}
```

此处在赋值时调用了隐式类型转换，若用户提供的类型已重载好类型转换则可以直接使用。

`decltype`是C++ 11加入的特性之一，可以根据表达式推导数据类型，有如此方便的自动类型推导，完成跨类型运算也算是信手拈来，不过需要注意的是用户提供的类型需要是可以推导的。

其实这里有想过要不要特例化`int*int`、`float*float`、`double*double`等，使用OpenBLAS进行运算，但思索了一下似乎有点跑题，而且需要用户安装OpenBLAS，徒然提升了使用门槛，就未作实现。

想自己动手？没问题！

本项目延续了一贯的高拓展性，实现了自定义一元/二元运算的框架，用户传入函数指针即可，而且二元运算也支持跨类型运算，即支持传入形如 `T3 foo(T1, T2)` 的函数指针。

```
//customized element-wise calculation
template<typename Tp>
template<typename result_type>
matrix<result_type> matrix<Tp>::unary_calc(result_type (*unary_function)(Tp))
const
{
    try
    {
        result_type *NewArr = new result_type[size()]{};
        size_t it = 0;
        for (matrix::channel_number ch = 1; ch <= channels; ch++)
        {
            for (size_t r = 1; r <= rows; r++)
            {
                for (size_t c = 1; c <= cols; c++)
                {
                    NewArr[it++] = unary_function(at(r, c, ch));
                }
            }
        }
        return matrix<result_type>(rows, cols, NewArr, channels);
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << size() << "\n";
        throw std::bad_alloc();
    }
}


template<typename Tp>
template<typename result_type, typename T2>
matrix<result_type> matrix<Tp>::binary_calc(const matrix<T2> &p, result_type
(*binary_function)(Tp, T2)) const
{
    if(cols!=p.get_cols()||rows!=p.get_rows()||channels!=p.get_channels())
    {
        throw std::invalid_argument(
                "Invalid Argument Exception: The size and channel number of two
matrix should be the same.\n");
    }
    try
    {
        result_type *NewArr = new result_type[size()]{};
        size_t it = 0;
        for (matrix::channel_number ch = 1; ch <= channels; ch++)
        {
            for (size_t r = 1; r <= rows; r++)
            {
```

```
                for (size_t c = 1; c <= cols; c++)
                {
                    NewArr[it++] = binary_function(at(r, c, ch), p.at(r, c,
ch));
                }
            }
        }
        return matrix<result_type>(rows, cols, NewArr, channels);
    }
    catch (std::bad_alloc &e)
    {
        std::cerr << "Bad Alloc Exception: Failed to allocate memory of the
given length " << size() << "\n";
        throw std::bad_alloc();
    }
}
```

其他功能

没有太多技术含量，包含了开发过程中为了方便而实现的中间产物等。

- data和matrix的硬拷贝：copy_to以及clone，总归是有用到它们的一天（
- data的重构中括号，==以及equals
- 矩阵的getters，因为不同模板类间私有成员不可见
- 矩阵的单个元素赋值set以及区域赋值fill，都可以轻易通过重载后的括号手动实现
- 矩阵输出流`<<`的重载
- 矩阵的转置

Part 3 - Result & Verification

本项目的测试用程序为`./src/benchmark.cpp`，以下测试结果均由矩阵计算器验证正确。

由于本项目并未使用跨平台时存在差异的内容，各组测试在开发板上运行结果一致，此处不做重复展示。

Testcase #1 创建矩阵，跨类型的乘法

三个通道的`4*4`整型矩阵`i_4_4_3`和三通道的`4*1 char`向量相乘

```
int *i_4_4_3 = new int[]
        {
                0, 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, 31,
                32, 33, 34, 35,
                36, 37, 38, 39,
                40, 41, 42, 43,
                44, 45, 46, 47
```

```
        };
char *uc_4_1_3 = new char[]
        {
                48, 49, 48, 49,
                49, 48, 49, 48,
                48, 48, 48, 48
        };

int main()
{
    matrix<int> i443(4, 4, i_4_4_3, 3);
    matrix<char> uc413(4, 1, uc_4_1_3, 3);
    cout << i443 << endl;
    cout << uc413 << endl;
    cout << i443 * uc413 << endl;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
{[0,    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,     31]
[32,    33,     34,     35;
36,     37,     38,     39;
40,     41,     42,     43;
44,     45,     46,     47]}
{[0;
1;
0;
1]
[1;
0;
1;
0]
[0;
0;
0;
0]}
{[292;
1068;
1844;
2620]
[3394;
4170;
4946;
5722]
[6432;
7200;
7968;
8736]}
```

Testcase #2 矩形ROI，软拷贝与内存管理

```
float *f = new float[]
        {
                0, 1, 2, 3, 4,
                5, 6, 7, 8, 9,
                10, 11, 12, 13, 14,
                15, 16, 17, 18, 19,
                20, 21, 22, 23, 24
        };
int main()
{
    matrix<float> f55(5, 5, f);
    matrix<float> sub(f55,2,2,4,4);
    //submatrix
    cout<<sub(2,2)<<endl;
    //modify the data shared by two matrices
    f55(3,3)=1.2;
    cout<<sub(2,2)<<endl;
    //delete the paraent matrix
    f55.~matrix();
    cout<<sub(2,2)<<endl;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ make && ./mat
Consolidate compiler generated dependencies of target mat
[ 50%] Building CXX object CMakeFiles/mat.dir/src/benchmark.cpp.o
[100%] Linking CXX executable mat
[100%] Built target mat
12
1.2
1.2
```

可以看到两个矩阵的确共用一块数据，并且在析构其中一个之后不影响另一个访问这块数据。

且经过在析构函数中输出ref_count检查，确认该项目不会存在内存的多次释放或是内存泄漏问题。

Testcase #3 掩膜ROI

```
float *f = new float[]
        {
                0, 1, 2, 3, 4,
                5, 6, 7, 8, 9,
                10, 11, 12, 13, 14,
                15, 16, 17, 18, 19,
                20, 21, 22, 23, 24
        };
int main()
{
    matrix<float> f55(5, 5, f);
    matrix<bool> b55(5,5,true);
    b55.fill(2,2,4,4, false);
    matrix<float> sub(5,5,6.6f);
    cout<<"before:\n"<<sub<<endl;
    f55.copy_to(sub,b55);
```

```
    cout<<"after:\n"<<sub<<endl;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
before:
[6.6,   6.6,    6.6,    6.6,    6.6;
6.6,    6.6,    6.6,    6.6,    6.6;
6.6,    6.6,    6.6,    6.6,    6.6;
6.6,    6.6,    6.6,    6.6,    6.6;
6.6,    6.6,    6.6,    6.6,    6.6]
after:
[0,     1,      2,      3,      4;
5,      0,      0,      0,      9;
10,     0,      0,      0,      14;
15,     0,      0,      0,      19;
20,     21,     22,     23,     24]
```

Testcase #4 通道拆分与自定义运算

```
int *i_4_4_3 = new int[]
        {
                0, 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, 31,
                32, 33, 34, 35,
                36, 37, 38, 39,
                40, 41, 42, 43,
                44, 45, 46, 47
        };
double HALF(int p)
{
    return p/2.0;
}
float DOUBLE(int p)
{
    return p*2.0f;
}
int main()
{
    matrix<int>main(4,4,i_4_4_3,3);
    matrix<float>double_red=main.split_channel(1).unary_calc(DOUBLE);
    matrix<int>green=main.split_channel(2);
    matrix<double>half_blue=main.split_channel(3).unary_calc(HALF);
    cout<<"double_red\n"<<double_red<<endl;
    cout<<"green\n"<<green<<endl;
    cout<<"half_blue\n"<<half_blue<<endl;
    matrix<double>sum=double_red+green+half_blue;
    cout<<"sum\n"<<sum<<endl;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
double_red
[0,     2,      4,      6;
8,      10,     12,     14;
16,     18,     20,     22;
24,     26,     28,     30]
green
[16,    17,     18,     19;
20,     21,     22,     23;
24,     25,     26,     27;
28,     29,     30,     31]
half_blue
[16,    16.5,   17,     17.5;
18,     18.5,   19,     19.5;
20,     20.5,   21,     21.5;
22,     22.5,   23,     23.5]
sum
[32,    35.5,   39,     42.5;
46,     49.5,   53,     56.5;
60,     63.5,   67,     70.5;
74,     77.5,   81,     84.5]
```

三种类型存储，红色加倍，绿色不变，蓝色减半，三者相加。

Testcase #5 自定义二元运算生成矩阵

```
int *i_4_4_3 = new int[]
        {
                0, 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, 31,
                32, 33, 34, 35,
                36, 37, 38, 39,
                40, 41, 42, 43,
                44, 45, 46, 47
        };

bool greater_than(int x, float y)
{
    return ((float) x) > y;
}

int main()
{
    float *f=new float[48]{};
    for(int i=0;i<48;i++)
    {
        f[i]=i%2?48.0:-1.0;
    }
```

```
    matrix<int> X(4, 4, i_4_4_3, 3);
    matrix<float> Y(4, 4, f,3);
    matrix<bool> R=X.binary_calc(Y, greater_than);
    cout<<R<<endl;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
{[1,    0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0]
[1,     0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0]
[1,     0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0;
1,      0,      1,      0]}
```

有两列Y>=X，其他X>Y

Testcase Bonus: 🎁

不知道从谁那里学来的方法(？)测试时去除了输出流的逗号和制表符:)

```
string *explore = new string[]
        {
                "Th", "mor", "earn ", "t C", "th", "re ig", "t I f",
                "Th", " you v", "muc", "for ", "r dedic", "eachin", "is cou"
        };

string *perseverance = new string[]
        {
                "e ", "e I l", "abou", "++, ", "e mo", "noran", "eel",
                "ank", "ery ", "h ", "you", "ation in t", "g th", "rse"
        };

#define CS_205 (CS + _205)
int main()
{
    matrix<string> CS(2, 7, explore);
    matrix<string> _205(2, 7, perseverance);
    cout << CS_205;
}
```

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
[The more I learn about C++, the more ignorant I feel;
Thank you very much for your dedication in teaching this course!]
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$
```

Part 4 - Difficulties & Solutions

Difficulty I 内存管理

题目要求避免硬拷贝，而且要做好内存管理，避免内存泄漏和多重删除。

Solution

本项目根据上课所述，使用了`ref_count`，`Step`等措施辅助内存管理，实现了上述目标，但根据实际应用场景，仍然实现了部分常用的硬拷贝函数，以避免对原数据进行修改导致多个矩阵的数据受到牵连。

Difficulty II 可拓展性

既然用了类模板，那么各种各样的跨类别、重载等问题就会如潮水一样涌来。

Solution

实际上我很清楚要完全做到这个库让人使用舒适，肯定是十分困难的事，因为用户可能的调用方法太多，有些难以用概括性(拓展性强)的写法容纳，所以只能尽可能地为用户提供自由度了，深切感谢`decltype`这一C++ 11的新特性，简直救人于水火之中。

比如项目中的跨类别赋值、运算、自定义比较与运算，都是经过反复调试和思考可能的调用方式后打补丁而成的。

Difficulty III OpenCV::mat?

前人的脚步已经走出太远，光是理解`mat.h`头文件的一部分就非常消耗时间和精力，而当实现ROI时，那种用户只需要给好参数，库会帮你解决一切的全能和自己写出来的完全是天壤之别。

Solution

这貌似不是一个短期内看上去可以解决的问题，要实现那样的效果，还缺少包括但不限于图论、几何等的前置知识，只能日后慢慢勤以补拙了，这次的可扩展性倒是让自己还算满意，非常自由。

希望以后也能加入到这样的一个项目里，为高楼大厦做一点微小的工作吧。

Part 5 - Summary

感谢您能读到这里。

关于本次项目的总结，言简意赅来讲就是把类模板**弄明白了又弄糊涂了**。

学了一学期C/C++，和上于老师的数据库一样，真的是越学越不会了(悲)，真的很喜欢这种上课风格。

一学期5个单人Projects，确实比大一感觉要繁忙许多，和朋友调侃的时候也总会提起自己这学期似乎一直“泡在Project里”。查了很多，翻了很多，熬了很多，但其实，学得还不够多。

在做Project的时候经常性地会把自己的项目和其他人的作比较，再进行完善，有时候会觉得好像没有必要做到这一步，但看到程序正常运行的时候还是会非常欣慰的，希望这样的感觉能稍微浸润到生活的其他方面吧。

最后的最后，诚挚感谢于老师、廖老师以及助教等为课程开展付出努力的人们！辛苦了！

```
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$ ./mat
[The more I learn about C++, the more ignorant I feel;
Thank you very much for your dedication in teaching this course!]
gutao@FIRST-MICROSOFT:/mnt/e/Cpp/Project05/build$
```

完结撒花 (○´∀`)ﾉ🌺