Project 5: A Class for Matrices

Name: 周思呈
SID: 12110644
code:

Part 01 - Analysis

用c++设计一个关于矩阵的类。Features如下：

1. 用 `variant` 处理多种数据类型。
2. 加入异常处理机制，用ERROR_TABLE宏和标准异常库处理异常。
3. 将矩阵头和数据储存位置拆分，并且加入ref_cnt指针记录引用次数，便于内存管理。
4. 设置默认构造参数，避免重复改写constructor。
5. 加入STEP变量实现ROI。
6. 重载各种不同数据类型之间运算符和输出流。

Part 02 - Code

数据储存方式

本次项目要求实现unsigned char, short, int, float, double五种数据类型。很直接的想法是使用模版类，但在实际实现过程中发现，模板类的使用存在以下几个问题：

1. 编译耗时长。模版类要求编译器在编译时将T替换成相应数据类型完成实例化，因此如果函数数量较多则会花费较长的编译时间。
2. 头文件长。为了用户使用方便，模版类写完之后可能需要在头文件中实例化并define别名，因此可能导致冗长的头文件，在复制粘贴过程中也可能出现较多错误。
3. 不同数据类型之间运算不便。在一个模板类中可能需要处理和另外五种不同数据类型的五种运算，也就是说对于同一个运算，可能至少需要写5+4+3+2+1个不同的函数来分别处理，这无异是容易出错又难以维护的。

考虑到上述原因，本次项目中本人避免了使用模板类，只使用了小部分模板函数来完成矩阵与不同类型常数之间的四则运算。为了处理五种不同的数据类型，最初我使用了union来储存不同类型的指针。这种实现方式有效地避免了不同数据类型之间运算难以有效实现的问题，但每次调用时都需要switch(TYPE)判断数据类型才能调用相应指针，在实现过程中耶十分繁琐。查阅了有关union的一些资料[1]后，我发现很多使用者都遇到了“switch case过于繁琐”这个问题，因此在c++17标准中出现了variant类。这个类和union非常类似，都是同一块地址可以储存不同几种数据类型，但同时只能储存一种数据类型，但其实现了visit函数让用户能够方便快捷地获取当前数据，避免了大篇幅switch case。因此本次项目中我采用variant储存五种不同类型的数据指针。

此外，我定义了Matrix专属的枚举类DataType表示其数据类型；SIZE变量用于储存该矩阵数据类型一个元素所占内存大小，使得比较两整数矩阵是否相同时可以直接使用memcmp函数而不用判断数据类型；ROW变量储存行数；COL变量储存列数；STEP变量用于实现ROI（在ROI部分中详细展开）；CHANNEL表示通道数；ref_cnt指针指向的地址储存这个矩阵被引用的次数，便于内存管理。

```
class Matrix {
public:
    enum DataType {
        TYPE8U,
        TYPE8S,
        TYPE4I,
        TYPE32F,
        TYPE64F
    };
    DataType TYPE;
    size_t SIZE;
    std::variant<unsigned char *, short *,
            int *, float *, double *> variant_pointer;
    size_t ROW;
    size_t COL;
    size_t STEP;
    int CHANNEL;
    size_t *ref_cnt;
    // ...
}
```

异常处理方式

在project3中我们用C实现了一个Matrix类，相比之下此项目使用的C++加入了异常处理机制，能够在出现错误时抛出异常提示错误信息，而非直接杀死程序，便于debug。在project4中我们用C实现了矩阵乘法的优化，重点关注性能提升，因此省略了一些参数检查和错误处理机制。相比之下本项目更需要关注与用户的交互，因此我在实现中用宏定义了异常处理机制，接收六种eid即错误编码，在std::err中打印错误提示信息及出错位置，实测debug非常好用。此外，有的函数涉及内存分配，可能出现动态分配失败的情况，因此在函数中使用ERROR_TABLE(eid)之前还加上了专门处理内存分配异常的模块。具体实现如下。

```
/*ERROR TABLE*/
#define ERROR_TABLE(eid)
if (eid == 1) std::cerr << "Illegal type in " << __FILE__
        << ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;\
else if (eid == 2) std::cerr << "Illegal size in " << __FILE__
```

```
        << ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;\
else if (eid == 3) std::cerr << "Index out of range in " << __FILE__
        << ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;\
else if (eid == 4) std::cerr << "Empty matrix in " << __FILE__
        << ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;\
else if (eid == 5) std::cerr << "Null pointer in " << __FILE__
        << ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;\
else std::cerr << "Unknown error in " << __FILE__ <<
        ": Line " << __LINE__ << " in function " << __FUNCTION__ << std::endl;
```

```
try{
        // do sth.
} catch (std::bad_alloc &ba) {
    std::cout << "bad_alloc exception!" << std::endl;
    std::cout << ba.what() << std::endl;
} catch (int eid) {
    ERROR_TABLE(eid)
}
```

内存管理方式

项目要求函数赋值时避免深拷贝，因此使用ref_cnt指针来储存一个矩阵的地址被引用了多少次。重载析构函数，每次将ref_cnt数量减1，在其数量变为0的时候才进行内存的释放。

```
bool Matrix::release(){
    visit([this](auto x){
        if (!x) {
            return false;
        }else {
            delete(x);
            delete(ref_cnt);
            return true;
        }
    }, variant_pointer);
    return true;
}

Matrix::~Matrix(){
    try{
        if (!ref_cnt) throw 5;
        (*ref_cnt)--;
        if (!(*ref_cnt)){ //ref_cnt==0释放内存
            if (!this->release()) throw 5;
        }
    }catch(int eid){
        ERROR_TABLE(eid)
    }
}
```

构造函数

重载了default constructor和copy constructor。
在重载默认构造函数过程中，为了避免复制粘贴，我在头文件声明构造函数时加入了默认构造参数，使用户调用构造函数时如果没有输入完整的参数列表也可以创建对象。默认构造参数row, col, channel均为1，防止在用户不输入这些参数时构造出空矩阵导致内存分配失败。当不输入指针或者输入的指针为空时，表示需要为指针分配内存空间，在判断数据类型后为相应指针申请相应大小动态内存。默认step为0，如果用户指定了step则按照用户指定的赋值，如果未指定则与列数相同。
copy constructor中先判断需要被拷贝的矩阵ROW, COL, CHANNEL和STEP是否为0，即是否为空矩阵，如果为空则抛出相应错误异常信息，如果非空就对类中的成员逐一赋值。

在Matrix.h文件中声明如下：

```
Matrix(DataType type = TYPE8U, size_t row = 1,
           size_t col = 1, void *dp = NULL, int channel = 1, size_t step = 0);
Matrix(const Matrix &m);
```

在Matrix.cpp文件中实现如下：

```
Matrix::Matrix(DataType type, size_t row, size_t col,
                           void* dp, int channel, size_t step) {
    try {
        //一通参数检查
        if (type > 4 || type < 0)
            throw 1;
        if (channel < 0)
            throw 2;
        if (row == 0 || col == 0)
            throw 4;

        TYPE = type;
        ROW = row;
        COL = col;
        CHANNEL = channel;
        if (step) STEP = step;
        else STEP = COL;
        ref_cnt = new size_t[sizeof(size_t)]{1};
        if (dp) { //传入非空指针
            switch (type) {
                case TYPE8U:
                    variant_pointer = (unsigned char *) dp;
                    SIZE = sizeof(unsigned char);
                    break;
                case TYPE8S:
                    variant_pointer = (short *) dp;
                    SIZE = sizeof(short);
                    break;
                case TYPE4I:
                    variant_pointer = (int *) dp;
                    SIZE = sizeof(int);
                    break;
                case TYPE32F:
                    variant_pointer = (float *) dp;
                    SIZE = sizeof(float);
                    break;
                case TYPE64F:
                    variant_pointer = (double *) dp;
                    SIZE = sizeof(double);
                    break;
                default:
                    throw 666;  //unknown error
            }
        } else { //指针为空，需要申请内存
            switch (type) {
                case TYPE8U:
                    dp = new unsigned char[row * col * channel
                            * sizeof(unsigned char)];
                    variant_pointer = (unsigned char *) dp;
                    SIZE = sizeof(unsigned char);
                    break;
                case TYPE8S:
                    dp = new short[row * col * channel * sizeof(short)];
                    variant_pointer = (short *) dp;
                    SIZE = sizeof(short);
                    break;
```

```
                case TYPE4I:
                    dp = new int[row * col * channel * sizeof(int)];
                    variant_pointer = (int *) dp;
                    SIZE = sizeof(int);
                    break;
                case TYPE32F:
                    dp = new float[row * col * channel * sizeof(float)];
                    variant_pointer = (float *) dp;
                    SIZE = sizeof(float);
                    break;
                case TYPE64F:
                    dp = new double[row * col * channel * sizeof(double)];
                    variant_pointer = (double *) dp;
                    SIZE = sizeof(double);
                    break;
                default:
                    std::cerr << "memory allocate fail" << std::endl;
                    throw 4;
            }
        }
    } catch (std::bad_alloc &ba) {
        std::cout << "bad_alloc exception!" << std::endl;
        std::cout << ba.what() << std::endl;
    } catch (int eid) {
            delete(ref_cnt);
        ERROR_TABLE(eid)
    }
}

Matrix::Matrix(const Matrix &m){
    try {
        //不存在空引用所以不用检查m是否为NULL
        if (m.ROW == 0 || m.COL == 0 || m.CHANNEL == 0 || m.STEP == 0) throw 4;
        if (!m.ref_cnt) throw 5;
        TYPE = m.TYPE;
        ROW = m.ROW;
        COL = m.COL;
        SIZE = m.SIZE;
        m.ref_cnt[0]++;
        ref_cnt = m.ref_cnt;
          visit([this](auto x){
            if (!x) throw 5;
            else variant_pointer = x;
        }, m.variant_pointer);
        variant_pointer = m.variant_pointer;
        CHANNEL = m.CHANNEL;
        STEP = m.STEP;
    }catch (int eid){
        ERROR_TABLE(eid)
    }
}
```

重载赋值和比较运算符

赋值运算符逻辑类似copy constructor，此处不再赘述。
比较运算符根据不同数据类型设计了两种不同比较方式。首先判断数据类型、ROW, COL, CHANNEL, STEP, SIZE是否完全相同，如果不同则判断为两个不同矩阵。unsigned char, short, integer均为整数类型，因此调用memcmp函数判断其所占内存是否完全相同。float和double均为浮点数类型，储存过程中存在一定误差，因此用visit函数获取variant对应类型指针，将对应元素作差取绝对值，判断是否小于对应类型的极小值，如果小于这个偏差就判断两数相等。由于使用一维矩阵储存数据，在进行取值判断的时候地址连续，相比二维更为高效。
相应使用`==`重载了`!=`。

```
Matrix &Matrix::operator=(const Matrix &m){
    try {
        if (m.ROW == 0 || m.COL == 0 || m.CHANNEL == 0 || m.STEP == 0) throw 4;
        if (!m.ref_cnt) throw 4;
        TYPE = m.TYPE;
        ROW = m.ROW;
        COL = m.COL;
        SIZE = m.SIZE;
        m.ref_cnt[0]++;
        ref_cnt = m.ref_cnt;
        visit([this](auto x){
            if (!x) throw 4;
            else variant_pointer = x;
        }, m.variant_pointer);
        CHANNEL = m.CHANNEL;
        STEP = m.STEP;
        return *this;
    }catch (int eid){
        ERROR_TABLE(eid)
    }
    return *this;
}

bool Matrix::operator==(const Matrix &m){
    if (TYPE != m.TYPE
        || ROW != m.ROW
        || COL != m.COL
        || CHANNEL != m.CHANNEL
        || STEP != m.STEP
        || SIZE != m.SIZE) {
        return false;
    }
    try{
        size_t size = ROW * COL * CHANNEL;
        switch (TYPE) {
            case TYPE8U:
            case TYPE8S:
            case TYPE4I:
                visit([&size, &m](auto x){
                    visit([&size, &x, &m](auto y){
                        if (!x || !y) throw 5;
                        if (!memcmp(x, y, size * m.SIZE))
                            return false;
                        else return true;
                    },m.variant_pointer);
                }, variant_pointer);
                break;
            case TYPE32F:
                visit([&size, &m](auto x){
                    visit([&size, &x, &m](auto y){
                        if (!x || !y) throw 5;
                        for (size_t i = 0; i < size; i++) {
                            if (abs(*(x + i) - *(y + i)) > FLT_MIN) return false;
                        }
                        return true;
                    },m.variant_pointer);
                }, variant_pointer);
                break;
            case TYPE64F:
                visit([&size, &m](auto x){
                    visit([&size, &x, &m](auto y){
                        if (!x || !y) throw 5;
                        for (size_t i = 0; i < size; i++) {
                            if (abs(*(x + i) - *(y + i)) > DBL_MIN) return false;
                        }
```

```
                        return true;
                    },m.variant_pointer);
                }, variant_pointer);
                break;
            default:
                throw 666;
        }
        return true;
    }catch (int eid){
        ERROR_TABLE(eid)
    }
    return false;
}

bool Matrix::operator!=(const Matrix &m){
    if (*this == m) return true;
    else return false;
}
```

重载矩阵与矩阵之间运算符

重载了`+` `-` `*`三种基本运算符，其中加法和减法逻辑与代码均类似，此处只阐述加法，减法与之同理。
因为涉及不同数据类型矩阵的运算，在进行运算之前需要先确定结果矩阵的类型。为了不丢失精度，结果矩阵的数据类型设定为进行运算的两个矩阵中精度较高的一个。判断方法为比较其枚举类TYPE数值大小，因为在定义该枚举类型时将其按精度排序，方便此处判断。确定结果矩阵类型之后再获取对应元素进行运算，此时最后得到的结果就是更高精度的数据类型。
获取相应元素的时候使用STEP取代COL以兼容ROI。
乘法实现逻辑大致类似，从project4中的优化方式上看，仅使用ikj算法加上omp并行就能获得良好的加速效果，运用指令集、命中cache等等优化方式并非此次poject重点，因此不再重复上一个项目中已经完成的工作。

```
Matrix Matrix::operator+(const Matrix & m) const
{
    try{
        if (ROW != m.ROW || COL != m.COL || CHANNEL != m.CHANNEL || STEP != m.STEP)
            throw 2;

        size_t c = COL * CHANNEL;
        DataType s_type = TYPE8U;
        if (TYPE <= m.TYPE){ //m的数据类型优先级更高
            s_type = m.TYPE;
        }else{
            s_type = TYPE;
        }
        Matrix sum(s_type, ROW, COL, NULL, CHANNEL, STEP);
        visit([&sum, &m, &c](auto x){
            visit([&sum, &x, &m, &c](auto y){
                visit([&x, &y, &m, &c](auto s){
                    if (!x || !y || !s) {
                            rsum.release();
                            throw 5;
                        }
                    for (size_t i = 0; i < m.ROW; i++){
                        size_t temp = i * m.STEP;
                        for (size_t j = 0; j < c; j++){
                            *(s + temp + j) = *(x + temp + j) + *(y + temp + j);
                        }
                    }
                }, sum.variant_pointer);
            },m.variant_pointer);
        }, variant_pointer);
        return sum;
    }catch(int eid){
```

```
        ERROR_TABLE(eid)
    }
    return *this;
}

Matrix Matrix::operator*(const Matrix & m) const
{
    try{
        if (ROW != m.COL || CHANNEL != m.CHANNEL)
            throw 2;
        size_t row = ROW;
        size_t col = m.COL;
        size_t mul = COL;
        DataType s_type = TYPE8U;
        if (TYPE <= m.TYPE){ //m的数据类型优先级更高
            s_type = m.TYPE;
        }else{
            s_type = TYPE;
        }
        Matrix sum(s_type, row, col, NULL, CHANNEL, m.STEP);
        visit([&row, &col, &mul, &sum, &m, this](auto x){
            visit([&row, &col, &mul, &sum, &x, &m, this](auto y){
                visit([&row, &col, &mul, &x, &y, &sum, &m, this](auto s){
                    if (!x || !y || !s) {
                            sum.release();
                            throw 4;
                        }
                    memset(s, 0, row * col * sum.SIZE);
#pragma omp parallel for
                    for (size_t i = 0; i < row; i++){
                        for (size_t k = 0; k < mul; k++){
                            for (size_t j = 0; j < col; j++){
                                *(s + i * m.STEP + j) +=
                                        *(x + i * STEP + k) * *(y + k * m.STEP + j);
                            }
                        }
                    }
                }, sum.variant_pointer);
            },m.variant_pointer);
        }, variant_pointer);
        return sum;
    }catch(int eid){
        ERROR_TABLE(eid)
    }
    return *this;
}
```

重载矩阵与常数之间运算符

Matrix类能够储存多种数据类型，与其进行运算的标量也有多重数据类型，因此采用模版类处理矩阵的标量运算。运算过程中结果矩阵的数据类型已经确定，因此能够隐式转换数据类型。同时，矩阵与标量的加减法满足交换律，因此使用友元函数处理c+m的情况。减法与加法逻辑类似，因此在减法的重载中使用了加法的重载，大大减少代码量。

```
template <typename T>
Matrix operator+(T c) const
{
    try{
        if (ROW == 0 || COL == 0 || CHANNEL == 0 || STEP == 0)
            throw 4;

        size_t size = ROW * COL * CHANNEL;
        Matrix sum(TYPE, ROW, COL, NULL, CHANNEL, STEP);
        visit([&size, &sum, &c, this](auto x){
            visit([&size, &sum, &x, &c, this](auto y){
```

```
                if (!x || !y) {
                    sum.release();
                    throw 4;
                }
                for (size_t i = 0; i < ROW; i++){
                    size_t temp = i * STEP;
                    for (size_t j = 0; j < c; j++){
                        *(y + temp + j) = *(x + temp + j) + c;
                    }
                }
            },sum.variant_pointer);
        }, variant_pointer);
        return sum;
    }catch(int eid){
        ERROR_TABLE(eid)
    }
    return *this;
}

template <typename T>
friend Matrix operator+(T c, const Matrix & m)
{
    return m+c;
}

template <typename T>
Matrix operator-(T c) const
{
    return *this + (-c);
}

template <typename T>
friend Matrix operator-(T c, const Matrix & m)
{
    return m-c;
}
```

同理重载乘法、乘法的友元函数、除法。

```
template <typename T>
Matrix operator*(T c) const
{
    try{
        if (ROW == 0 || COL == 0 || CHANNEL == 0 || STEP == 0)
            throw 4;

        size_t size = ROW * COL * CHANNEL;
        Matrix sum(TYPE, ROW, COL, NULL, CHANNEL, STEP);
        visit([&size, &sum, &c, this](auto x){
            visit([&size, &sum, &x, &c, this](auto y){
                if (!x || !y) {
                    sum.release();
                    throw 4;
                }
                for (size_t i = 0; i < ROW; i++){
                    size_t temp = i * STEP;
                    for (size_t j = 0; j < c; j++){
                        *(y + temp + j) = *(x + temp + j) * c;
                    }
                }
            },sum.variant_pointer);
        }, variant_pointer);
        return sum;
    }catch(int eid){
```

```
        ERROR_TABLE(eid)
    }
    return *this;
}

template <typename T>
friend Matrix operator*(T c, const Matrix & m)
{
    return m * c;
}

template <typename T>
Matrix operator/(T c) const
{
    return (*this) * (1/double(c));
}
```

重载输出运算符

重载了输出矩阵的运算符和输出矩阵类型的`<<`运算符。由于需要输出的矩阵可能很大，重载时对于大矩阵采用了省略的打印方式，即当行数或者列数超过16时只打印其前4行/列信息并打印总共行/列数，而对于规模较小的矩阵则将其全部输出。
重载输出矩阵类型运算符，将枚举类型对应的类型名称打印出来。

```
std::ostream & operator<<(std::ostream & os, const Matrix & m)
{
    try {
        size_t c = m.COL * m.CHANNEL;
        visit([&c, &os, &m](auto x) {
            if (!x) throw 4;
            if (c >= 16 && m.ROW >= 16) {
                os << "total " << m.ROW << " rows and "
                        << c << " cols" << std::endl;
                for (int i = 0; i < 4; i++) { //前四行
                    for (int j = 0; j < 4; j++) { //前四列
                        os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                << std::left << *(x + i * m.STEP + j);
                    }
                    os << "......  ";
                    for (size_t j = c-4; j < c; j++) { //后四列
                        os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                << std::left << *(x + i * m.STEP + j);
                    }
                    os << std::endl;
                }
                os << "......" << std::endl;
                for (int i = m.ROW-4; i < m.ROW; i++) { //后四行
                    for (int j = 0; j < 4; j++) { //前四列
                        os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                << std::left << *(x + i * m.STEP + j);
                    }
                    os << "......  ";
                    for (size_t j = c-4; j < c; j++) { //后四列
                        os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                << std::left << *(x + i * m.STEP + j);
                    }
                    os << std::endl;
                }
            }else if (c >= 16){
                os << "total " << c << " cols" << std::endl;
                for (size_t i = 0; i < m.ROW; i++){
                    for (int j = 0; j < 4; j++) { //前四列
                        os << std::setprecision(PRECISION) << std::setw(WIDTH)
```

```
                                << std::left << *(x + i * m.STEP + j);
                        }
                        os << "......  ";
                        for (size_t j = c-4; j < c; j++) { //后四列
                            os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                    << std::left << *(x + i * m.STEP + j);
                        }
                        os << std::endl;
                    }
                }else if (m.ROW >= 16){
                    os << "total " << m.ROW << " rows" << std::endl;
                    for (size_t i = 0; i < 4; i++){
                        for (size_t j = 0; j < c; j++){
                            os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                    << std::left << *(x + i * m.STEP + j);
                        }
                        os << std::endl;
                    }
                    os << "......" << std::endl;
                    for (size_t i = m.ROW-4; i < m.ROW; i++){
                        for (size_t j = 0; j < c; j++){
                            os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                    << std::left << *(x + i * m.STEP + j);
                        }
                        os << std::endl;
                    }
                }else{
                    for (size_t i = 0; i < m.ROW; i++){
                        for (size_t j = 0; j < c; j++){
                            os << std::setprecision(PRECISION) << std::setw(WIDTH)
                                    << std::left << *(x + i * m.STEP + j);
                        }
                        os << std::endl;
                    }
                }
            }, m.variant_pointer);
            return os;
        }catch (int eid){
            ERROR_TABLE(eid)
        }
        return os;
    }

    std::ostream & operator<<(std::ostream & os, const Matrix::DataType &tp)
    {
        switch (tp){
            case Matrix::TYPE8U:
                os << "TYPE8U";
                break;
            case Matrix::TYPE64F:
                os << "TYPE64F";
                break;
            case Matrix::TYPE8S:
                os << "TYPE8S";
                break;
            case Matrix::TYPE32F:
                os << "TYPE32F";
                break;
            case Matrix::TYPE4I:
                os << "TYPE4I";
                break;
            default:
                break;
        }
        return os;
    }
```

设置和获取指定位置元素

与标量运算相同，要获取和指定位置元素的类型可能也有很多种，因此仍然采用模版类进行实现。setElement所设置的数据类型由输入参数的数据类型决定，getElement获取的数据类型由接收数据的变量类型决定。在对矩阵元素赋值和获取矩阵元素时进行隐式类型转换。

```
template <typename T>
bool setElement(size_t r, size_t c, int channel, T value) {
    try {
        if (r >= ROW || c >= COL || channel > CHANNEL)
            throw 3;
        size_t index = r * CHANNEL * STEP + c * channel;
        visit([&index, &value](auto x){
            if (!x) throw 4;
            *(x + index) = value;
        }, variant_pointer);
        return true;
    } catch (int eid) {
        ERROR_TABLE(eid)
    }
    return false;
}

template <typename T>
bool getElement(size_t r, size_t c, int channel, T &value) {
    try {
        if (r >= ROW || c >= COL || channel > CHANNEL)
            throw 3;
        size_t index = r * CHANNEL * STEP + c * channel;
        visit([&index, &value](auto x){
            if (!x) throw 4;
            value = *(x + index);
        }, variant_pointer);
        return true;
    } catch (int eid) {
        ERROR_TABLE(eid)
    }
    return false;
}
```

ROI

输入原始矩阵、结果矩阵、需要获取的矩阵左上角地址、需要获取矩阵大小，返回值表示是否获取成功。
首先进行参数合法性检查，如果需要获取的矩阵边界超出原矩阵边界则抛出异常。参数检查之后对结果矩阵成员进行赋值，将结果矩阵数据指针指向需要获取的矩阵左上角，然后将原始矩阵引用数量加一，避免内存拷贝，方便内存管理。

```
bool ROI(const Matrix& original, Matrix &result, size_t indexR,
                size_t indexC, size_t rows, size_t cols){
    try{
        if (original.ROW < indexR || indexR + rows >= original.ROW
        || original.COL < indexC || indexC + cols >= original.COL)
            throw 3;
        result.STEP = original.STEP;
        result.SIZE = original.SIZE;
        result.TYPE = original.TYPE;
        result.CHANNEL = original.CHANNEL;
        result.COL = cols;
        result.ROW = rows;
        size_t bias = original.COL * original.CHANNEL * indexR + indexC;
        visit([bias, &result](auto x){
            if (!x) throw 4;
```

```
            result.variant_pointer = x+bias;
        }, original.variant_pointer);
        if (!original.ref_cnt) throw 5;
        else (*original.ref_cnt)++;
        if (result.ref_cnt) delete(result.ref_cnt);
        result.ref_cnt = original.ref_cnt;
        return true;
    }catch(int eid){
        ERROR_TABLE(eid)
    }
    return false;
}
```

从文件中读取数据

为了方便大规模矩阵测试，实现了从文件中读取数据的方法。输入需要读取的文件名，指定数据类型，指定要读取的数据规模，返回对应类型指针。在这里可以看到，由于数据类型未知，在创建指针申请相应内存之前需要先switch case数据类型，导致复制粘贴代码较多。原本如果在矩阵的数据存储上使用union也会出现类似的情况，十分繁琐，幸运的是标准库提供的variant解决了这个问题。

```
void* ReadFromFile(std::string filename, Matrix::DataType type, size_t size){
    try {
        char data[100]; //字符串
        int index = 0;
        // 以读模式打开文件
        std::ifstream infile;
        infile.open(filename);
        switch (type) {
            case Matrix::TYPE8U: {
                unsigned char *dp = new unsigned char[size * sizeof(unsigned char)];
                while (infile >> data && size > index) {
                    *(dp + index) = (unsigned char) (std::stoi(data));
                    index++;
                }
                infile.close();
                return dp;
            }
            case Matrix::TYPE8S: {
                short *dp = new short[size * sizeof(short)];
                while (infile >> data && size > index) {
                    *(dp + index) = short(std::stoi(data));
                    index++;
                }
                infile.close();
                return dp;
            }
            case Matrix::TYPE4I: {
                int *dp = new int[size * sizeof(int)];
                while (infile >> data && size > index) {
                    *(dp + index) = std::stoi(data);
                    index++;
                }
                infile.close();
                return dp;
            }
            case Matrix::TYPE32F:{
                float *dp = new float[size * sizeof(float)];
                while (infile >> data && size > index) {
                    *(dp + index) = std::stof(data);
                    index++;
                }
                infile.close();
                return dp;
```

```
            }
            case Matrix::TYPE64F:{
                double *dp = new double[size * sizeof(double)];
                while (infile >> data && size > index) {
                    *(dp + index) = std::stod(data);
                    index++;
                }
                infile.close();
                return dp;
            }
            default:
                throw 666;
        }
    }catch (std::bad_alloc & ba) {
        std::cout << "bad_alloc exception!" << std::endl;
        std::cout << ba.what() << std::endl;
    }catch (int eid){
        ERROR_TABLE(eid)
    }
}
```

Part 03 - Result & Verifification

异常处理示例

测试代码：

```
int* ip = new int[12 * sizeof(int)]{1,1,1,1,
                                    2,2,2,2,
                                    3,3,3,3};
Matrix matrix(Matrix::TYPE4I, 3, 4, ip,1);
int i = 0;
matrix.getElement(6,3,1,i);
matrix.setElement(3,6,1,i);
```

测试结果：

```
(base) zhousicheng@zhousichengdeMacBook-Pro project_5_matrixCPP % ./a.out
Index out of range in ./Matrix.h: Line 213 in function getElement
Index out of range in ./Matrix.h: Line 196 in function setElement
```

重载输出运算符结果

测试代码（afile.dat文件里放的是从1到32*32的数）：

```
Matrix mt(Matrix::TYPE4I, 32, 32, ReadFromFile("afile.dat",Matrix::TYPE4I, 1024));
cout << mt << endl;

Matrix mt1(Matrix::TYPE4I, 32, 8, ReadFromFile("afile.dat",Matrix::TYPE4I, 32*8));
cout << mt1 << endl;

Matrix mt2(Matrix::TYPE4I, 8, 32, ReadFromFile("afile.dat",Matrix::TYPE4I, 32*8));
cout << mt2 << endl;
```

测试结果：

```
total 32 rows and 32 cols
0    1    2    3    ......  28   29   30   31
32   33   34   35   ......  60   61   62   63
64   65   66   67   ......  92   93   94   95
96   97   98   99   ......  124  125  126  127
......
896  897  898  899  ......  924  925  926  927
928  929  930  931  ......  956  957  958  959
960  961  962  963  ......  988  989  990  991
992  993  994  995  ......  1020 1021 1022 1023
```

```
total 32 rows
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
......
224  225  226  227  228  229  230  231
232  233  234  235  236  237  238  239
240  241  242  243  244  245  246  247
248  249  250  251  252  253  254  255

total 32 cols
0    1    2    3    ......  28   29   30   31
32   33   34   35   ......  60   61   62   63
64   65   66   67   ......  92   93   94   95
96   97   98   99   ......  124  125  126  127
128  129  130  131  ......  156  157  158  159
160  161  162  163  ......  188  189  190  191
192  193  194  195  ......  220  221  222  223
224  225  226  227  ......  252  253  254  255
```

ROI测试结果

测试代码：

```
Matrix mt(Matrix::TYPE4I, 32, 32, ReadFromFile("afile.dat",Matrix::TYPE4I, 1024));
cout << mt << endl;

ROI(mt, mtt, 5, 5, 2, 2);
cout << mtt << endl;
```

测试结果：

```
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
48   49   50   51   52   53   54   55
56   57   58   59   60   61   62   63

45   46
53   54
```

不同类型矩阵运算

测试代码：

```
int* ip = new int[12 * sizeof(int)]{1,1,1,1,
                                    2,2,2,2,
                                    3,3,3,3};
    double* dp = new double[12 * sizeof(double)]{1.1,2.2,3.3,4.4,
                                                 5.5,6.6,7.7,8.8,
                                                 9.9,1.1,2.2,3.3};
    Matrix matrix(Matrix::TYPE4I, 3, 4, ip,1);
    Matrix matrix2(Matrix::TYPE64F, 4, 3, dp,1);
    Matrix mt = matrix * matrix2;
    cout << matrix << endl;
    cout << matrix2 << endl;
    cout << mt << endl;
    cout << mt.TYPE << endl;
```

测试结果：

```
1     1     1     1
2     2     2     2
3     3     3     3

1.1   2.2   3.3
4.4   5.5   6.6
7.7   8.8   9.9
1.1   2.2   3.3

14.3 18.7 23.1
28.6 37.4 46.2
42.9 56.1 69.3

TYPE64F
```

矩阵与常数之间运算

测试代码：

```
int* ip = new int[12 * sizeof(int)]{1,1,1,1,
                                    2,2,2,2,
                                    3,3,3,3};
Matrix matrix(Matrix::TYPE4I, 3, 4, ip,1);
cout << matrix+1 << endl;
cout << matrix-1 << endl;
```

测试结果：

```
(base) zhousicheng@zhousichengdeMacBook-Pro project_5_matrixCPP % ./a.out
2    2    2    2
3    3    3    3
4    4    4    4

0    0    0    0
1    1    1    1
2    2    2    2
```

设置和获取指定位置元素

测试代码：

```
Matrix mt(Matrix::TYPE4I, 8, 8, ReadFromFile("afile.dat",Matrix::TYPE4I, 64));
cout << mt << endl;
mt.setElement(3,3,1,6.66);
cout << mt << endl;
float i;
mt.getElement(3,3,1,i);
cout << "i = " << i << endl;
```

测试结果：

```
(base) zhousicheng@zhousichengdeMacBook-Pro project_5_matrixCPP % ./a.out
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
48   49   50   51   52   53   54   55
56   57   58   59   60   61   62   63

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   6    28   29   30   31
32   33   34   35   36   37   38   39
40   41   42   43   44   45   46   47
48   49   50   51   52   53   54   55
56   57   58   59   60   61   62   63

i = 6
```

Part 04 - Difficulties & Solutions

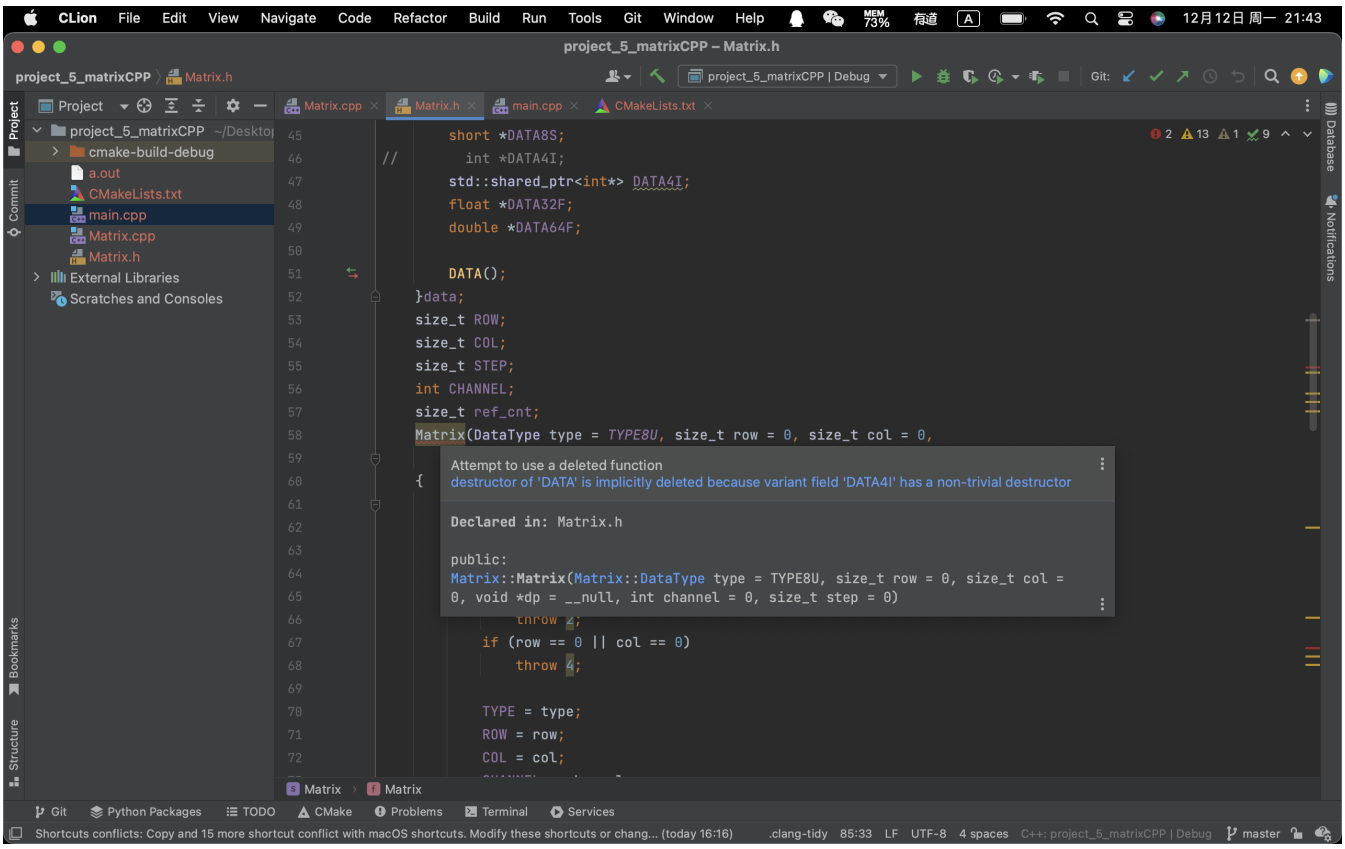
union和shared_ptr

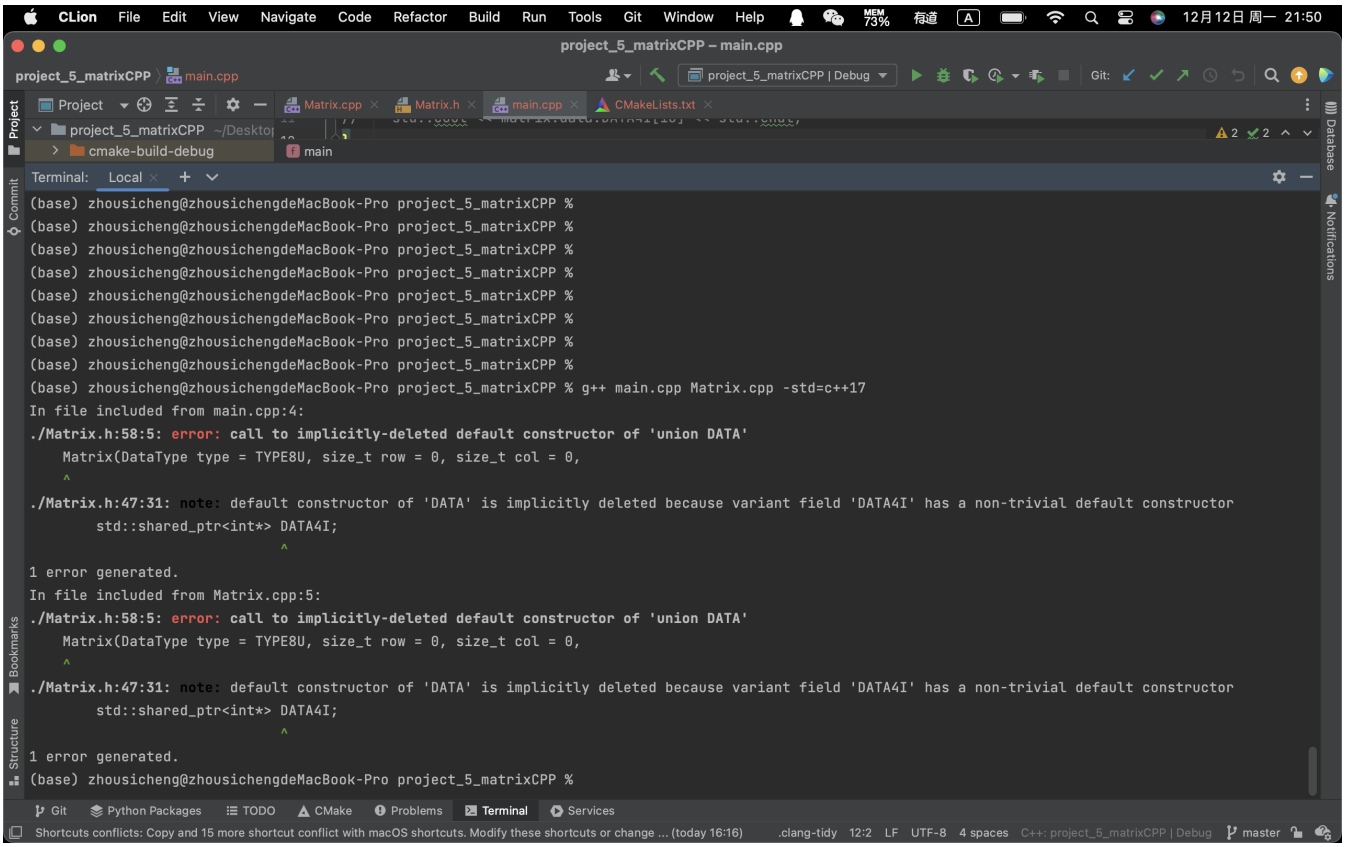
在最初的计划中，我打算使用union储存数组指针，用shared_ptr进行内存管理。这就需要将union中的成员设为shared_ptr。但实际操作中发现这并不可行。

在成员中定义如下：

```
31  class Matrix {
32  public:
33      enum DataType{...};
40      DataType TYPE;
41      union DATA
42      {
43          void * DATANULL;
44          unsigned char * DATA8U;
45          short *DATA8S;
46  //          int *DATA4I;
47          std::shared_ptr<int*> DATA4I;
48          float *DATA32F;
49          double *DATA64F;
50
51          DATA();
52      }data;
```

报错信息如下：

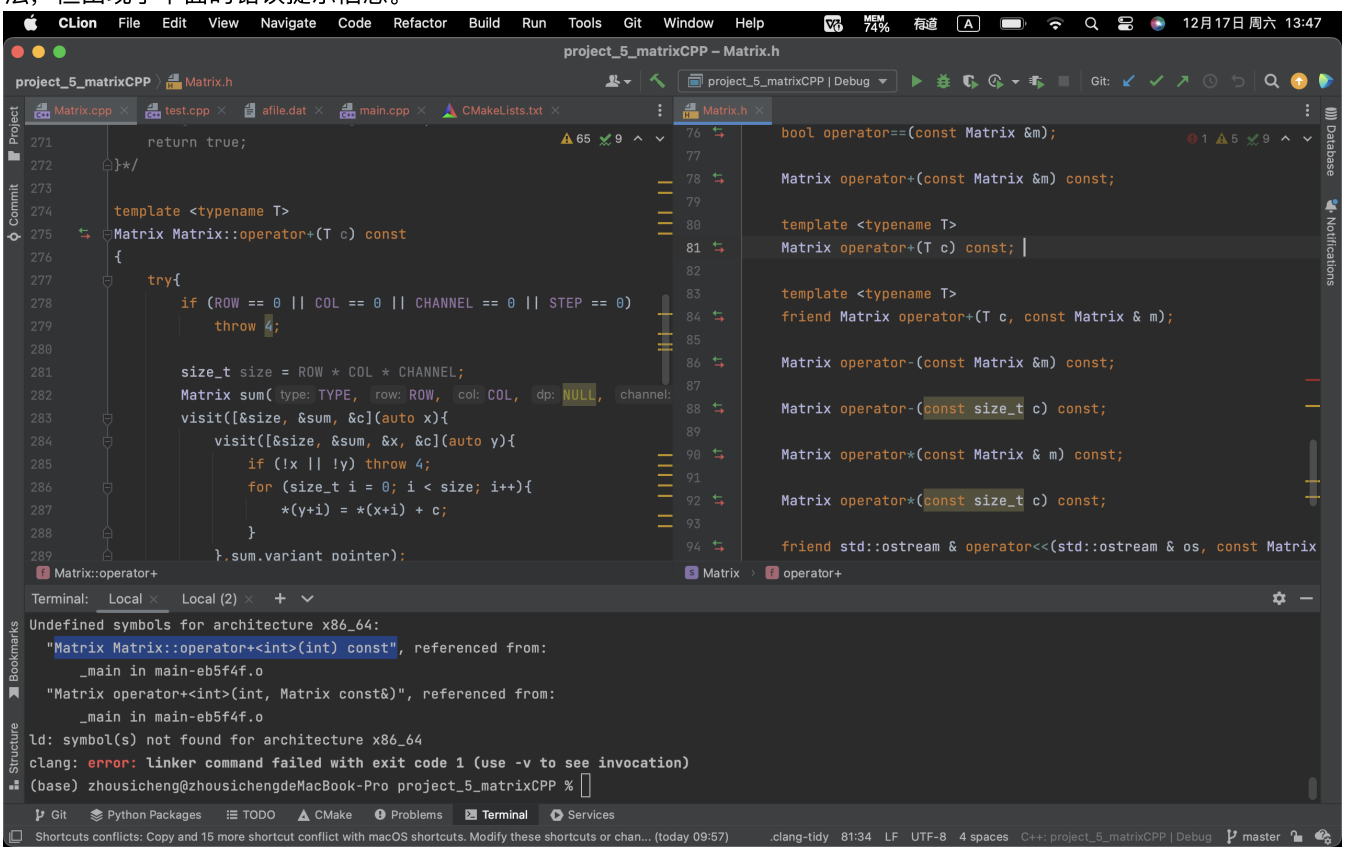




根据clang-tidy和编译器提示尝试了多种修改方法，但都没能完全消除报错信息。猜想是因为shared_ptr对象在union中出现时需要实例化，但union中同时只能存在一个数据，因此编译器不知道什么时候去调用智能指针对应的构造函数和析构函数，因此不能这样使用。
在之后的数据储存中使用了variant代替union，使用了计数器ref_cnt代替智能指针。

模版函数放置位置

通常在实现过程中会把函数的声明放在.h文件中、实现放在.cpp文件中。对于模板函数，一开始我也采用了这样的写法，但出现了下面的错误提示信息。



错误提示说找不到这个函数，可知并未编译成功。查资料得知，C++中每一个对象所占用的空间大小在编译时确定，在模板类被实例化之前，编译器无法知道模板类中使用模板类型的对象所占用的空间大小[2]。解决方法是将模板函数声明和定义均写入.h文件。

---

1. https://en.cppreference.com/w/cpp/language/union ↩
2. # 【C++】模板函数的声明和定义必须在同一个文件中 http://t.csdn.cn/htYDm ↩