CS205 C/C++ Programming -- Project 4 A Class to Describe a Matrix

Name: 陈长信(Chen Changxin)

SID: 12210731

Part1--问题分析

本项目旨在建造Matrix类来存储矩阵以及指向数据块的指针，并通过对运算符的重载对不同数据类型的矩阵完成加减乘赋值等基本运算。

在project3关于矩阵乘法加速的探究并没有在此项目进行融合，本项目侧重于对矩阵存储方式的探究以及优化策略。

避免使用hard_copy对矩阵进行赋值，使用内存共享的思路：

1. 避免使用硬拷贝，这里使用软拷贝，省略了ref_count的计数，智能指针的std实现内部相当于已经有count计数，用std::shared_ptr对内存空间进行优化管理，避免冗余的内存空间开辟。
2. 使用opencv中ROI的思路对内存进行优化，即只对感兴趣的部分进行处理，相当于是对内存进行共用，用同一块的内存，然后不同的矩阵占据内存块的相应部分。

Part2--代码结构解析

main()函数测试集

1. main()函数主要针对Class Matrix中符号的重载进行测试，以short类型的矩阵类型举例子（int,float,long,double等同理），通过创建shared_ptr智能指针来实现内存空间的共用，这样避免在后续assign时候的偏离导致在内存空间销毁的时候无法清除目标地址的数据。

```
std::shared_ptr<short[]> A_short_data(new short[n * n]);
Matrix<short> A_short(n, n, A_short_data);
auto B_short = A_short;
auto C_short = A_short;
auto D_short = A_short;
```

2. 对不同数据类型的数据通过rand()进行赋值，为后续计算做基础。

```
std::shared_ptr<short[]> A_short_data(new short[n * n]);
Matrix<short> A_short(n, n, A_short_data);
auto B_short = generateRandomMatrix<short>(n, n);

std::shared_ptr<int[]> A_int_data(new int[n * n]);
Matrix<int> A_int(n, n, A_int_data);

std::shared_ptr<float[]> A_float_data(new float[n * n]);
Matrix<float> A_float(n, n, A_float_data);

std::shared_ptr<double[]> A_double_data(new double[n * n]);
Matrix<double> A_double(n, n A_double_data);
```

3. 对矩阵进行运算符的测试（这里以+进行举例）

```
std::cout << "Testing addition operator (+)..." << std::endl;
auto result_add_short = A_short + B_short;
auto result_add_int = A_int + B_int;
auto result_add_float = A_float + B_float;
auto result_add_double = A_double + B_double;
```

Matrix类中代码结构

1. **声明类中的成员变量。**其中size_t startRow, size_t startCol, size_t roiRows, size_t roiCols代表的是他data存储空间内实际上有效的数据区域范围，这样的ROI方法有利于内存的优化，在赋值方法的时候不需要创建新的矩阵，只需要在创建或者复制的时候告诉他需要的区域参数就好，另外这里使用`std::shared_ptr<T[]>data`来存储数据。

```
public:
    size_t rows;
    size_t cols;
    std::shared_ptr<T[]> data;
//这里后面四个变量表示对当前数据空间内具体数据的位置标记
    size_t startRow;
    size_t startCol;
    size_t roiRows;
    size_t roiCols;
    size_t size;
```

2. **重写constructor构造函数。**这里为了使用的方便使用两个construtor，第一个构造函数允许直接创建一个新的矩阵对象，并为其分配内存空间。而第二个构造函数则允许使用现有的数据数组来初始化矩阵对象，因为当已经有一个已分配的内存块，并且想要将其用作矩阵的数据存储可以共享内存空间实现优化。

```
Matrix(size_t rows, size_t cols, size_t startRow, size_t startCol, size_t roiRows,
size_t roiCols) : rows(rows), cols(cols), startRow(startRow), startCol(startCol),
roiRows(roiRows), roiCols(roiCols)
{
        data = std::shared_ptr<T[]>(new T[rows * cols]);
}

// Constructor with shared pointer to data
Matrix(size_t rows, size_t cols, std::shared_ptr<T[]> data, size_t startRow, size_t
startCol, size_t roiRows, size_t roiCols) : rows(rows), cols(cols),
startRow(startRow), startCol(startCol), roiRows(roiRows), roiCols(roiCols),
data(data)
{}
```

3. **使用默认的destructor析构函数。**

```
~Matrix() = default;
```

4. **赋值号操作符的重写。**如果是同样的实体矩阵就直接return，否则就进行赋值，注意这里的 `data = other.data` 并没有开辟额外的内存空间，而是相当于复用了shared_ptr中同样的一段内存空间，然后内部的ref_cnt++。

```
Matrix &operator=(const Matrix &other)
{
    if (this != &other)
    {
        rows = other.rows;
        cols = other.cols;
        data = other.data;//自动增加计数
     }
     return *this;
}
```

5. **对其他包括"==","+","-","*"等符号的重写。**创建result矩阵(rows,cols)来存储结果矩阵，最后返回，然后最后进行打印。

```
// Equality operator
bool operator==(const Matrix &other) const
{
    if (rows != other.rows || cols != other.cols)
        return false;
    for (size_t i = 0; i < rows * cols; ++i)
    {
        if (data[i] != other.data[i])
            return false;
    }
    return true;
```

```
}

// Addition operator
Matrix operator+(const Matrix &other) const
{
    if (rows != other.rows || cols != other.cols)
    throw std::invalid_argument("Matrices in different dimension for addition");
    Matrix result(rows, cols, startRow, startCol, roiRows, roiCols);
    for (size_t i = 0; i < rows * cols; ++i)
    {
        result.data[i] = data[i] + other.data[i];
    }
    return result;
}

// Subtraction operator
Matrix operator-(const Matrix &other) const
{
    if (rows != other.rows || cols != other.cols)
        throw std::invalid_argument("Matrices in different dimension for
subtraction");
    Matrix result(rows, other.cols, startRow, startCol, roiRows, roiCols);
    for (size_t i = 0; i < rows * cols; ++i)
    {
        result.data[i] = data[i] - other.data[i];
    }
    return result;
}

// Multiplication operator
Matrix operator*(const Matrix &other) const
{
    if (cols != other.rows)
        throw std::invalid_argument("Invalid multiplication!");
    Matrix result(rows, other.cols, startRow, startCol, roiRows, roiCols);
    for (size_t i = 0; i < rows; ++i)
    {
        for (size_t j = 0; j < other.cols; ++j)
        {
            T sum = 0;
            for (size_t k = 0; k < cols; ++k)
            {
                sum += (*this)(i, k) * other(k, j);
            }
            result(i, j) = sum;
        }
     }
     return result;
}
```

6. 对()进行符号重写，这样方便对矩阵空间内某个元素的快速访问。

```
// Element access
T &operator()(size_t i, size_t j)
{
    if (i ⩾ startRow && i < startRow + roiRows && j ⩾ startCol && j < startCol +
roiCols)
    {
        return data[(i - startRow) * cols + (j - startCol)];
    }
    else
    {
        throw std::out_of_range("Index out of range");
    }
}
```

其他函数

1. `generate` 函数，对不同类型的矩阵进行随机值赋用。这里将整数类型与小数类型分开进行random赋值，因为如果合在一起进行统一化赋值容易生成全是0的int类型矩阵。

```
template <typename T>
Matrix<T> generateRandomMatrix(size_t rows, size_t cols)
{
    Matrix<T> mat(rows, cols, 0, 0, rows, cols);
    if constexpr (std::is_floating_point<T>::value)
    {
        // For floating-point types.
        for (size_t i = 0; i < rows; ++i)
        {
            for (size_t j = 0; j < cols; ++j)
            {
                mat(i, j) = static_cast<T>(rand()) / static_cast<T>(RAND_MAX); //
Random float between 0 and 1
            }
        }
    }
    else
    {
        // For integer types.
        for (size_t i = 0; i < rows; ++i)
        {
            for (size_t j = 0; j < cols; ++j)
            {
                mat(i, j) = static_cast<T>(rand() % 100); // Random integer between 0
and RAND_MAX
            }
        }
    }
```

```
    return mat;
}
```

2. `printMatrix` 函数，对不同类型的矩阵均可以进行打印，方便对结果的验证以及对比。

```
template <typename T>
void printMatrix(const Matrix<T> &mat)
{
    for (size_t i = mat.startRow; i < mat.roiRows; ++i)
    {
        for (size_t j = mat.startCol; j < mat.roiCols; ++j)
        {
            std::cout << std::setw(4) << mat(i, j) << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

Part3--结果测试

本部分进行main函数中进行的符号重写测试结果的展示。**出于篇幅以及简洁性的考虑，这里仅展示对int类型的3*3矩阵运算结果。**

具体测试代码如下

```
std::shared_ptr<int[]> A_int_data(new int[size * size]);
Matrix<int> A_int(size, size, A_int_data, 0, 0, size, size);
auto B_int = A_int;
auto C_int = A_int;
auto D_int = A_int;


A_int = generateRandomMatrix<int>(size, size);
B_int = generateRandomMatrix<int>(size, size);
C_int = generateRandomMatrix<int>(size, size);
D_int = generateRandomMatrix<int>(size, size);

//printout all the random Matrix

// Test assignment operator
std::cout << "Testing assignment operator (=)..." << std::endl;
A_int = B_int;
std::cout << "Matrix A after assignment:" << std::endl;
printMatrix(A_int);
```

```
// Test equality operator
std::cout << "Testing equality operator (==)..." << std::endl;
std::cout << "A_int == B_int: " << std::boolalpha << (A_int == B_int) << std::endl;

auto result_add_int = A_int + B_int;

auto result_sub_int = A_int - C_int;

auto result_mul_int = A_int * D_int;

//Test ROI implementation
std::cout << "Testing assignment with different ROI..." << std::endl;
Matrix<int> D_int(size, size, data, 0, 0, 2, 2); // Create a matrix with different
ROI(0,0,2,2) just as an ROI area example, represents D_int only care about the ROI
area
D_int = C_int;                              // Assign C_int to D_int
std::cout << "Matrix D after assignment with different ROI:" << std::endl;
printMatrix(D_int);

//Test For Different types(Let's take short for example)
printf("Test for different types:\n");
std::shared_ptr<short[]> A_short_data(new short[n * n]);
Matrix<short> A_short(n, n, A_short_data, 0, 0, n, n);
A_short = generateRandomMatrix<short>(n, n);
auto B_short = A_short;
std::cout << "Matrix B_short:" << std::endl;
printMatrix(B_short);
std::cout << "sum_short:" << std::endl;
printMatrix(A_short + B_short);
```

结果如下图所示（以size = 3举例）

```
daniel@Daniel-Chen:~/YU++/Project4-Matagain$ g++ Matrix.cpp
daniel@Daniel-Chen:~/YU++/Project4-Matagain$ ./a.out
Please input size: 3
Matrix A (int):
  42   51   43
  63   82   83
  62   30   88
Matrix B (int):
  77   61   94
  31   49   71
  81   47   48
Matrix C (int):
  20   92   47
  92   34   89
  47   93   35
Testing assignment operator (=)...
Matrix A after assignment:
  77   61   94
  31   49   71
  81   47   48
Testing equality operator (==)...
A_int == B_int: true
Testing addition operator (+)...
Result of addition:
 154  122  188
  62   98  142
 162   94   96
Testing subtraction operator (-)...
Result of subtraction:
  57  -31   47
 -61   15  -18
  34  -46   13
Testing multiplication operator (*)...
Result of multiplication:
11570 17900 12338
8465 11121 8303
8200 13514 9670
Testing assignment with different ROI...
Matrix D after assignment with different ROI:
  20   92
  92   34
```

完成对符号重写的检测

```
daniel@Daniel-Chen:~/YU++/Project4-Matagain$ ./a.out
Please input size: 3
Test for different types:
Matrix B_short:
  53   32   26
  13    0    8
  55    1   27

sum_short:
 106   64   52
  26    0   16
 110    2   54

Matrix B_float:
0.106787 0.63489 0.0939144
0.746051 0.68532 0.884703
0.833332 0.934372 0.47747

sum_float:
0.213575 1.26978 0.187829
1.4921 1.37064 1.76941
1.66666 1.86874 0.954941

Matrix B_double:
0.563844 0.71839 0.48614
0.978908 0.571187 0.581697
0.0261796 0.2932 0.891849

sum_double:
1.12769 1.43678 0.97228
1.95782 1.14237 1.16339
0.0523593 0.586399 1.7837
```

完成对不同数据类型矩阵赋值以及简单运算的检验

Part4--项目总结

1. 本项目试图通过对矩阵类的模拟对矩阵的主要参数进行存储，以对符号重载的方式完成对矩阵的简易操作同时完成了对内存管理的优化方案。

Part5--补充说明

1. 本项目部分代码框架来自ChatGPT，核心代码均为本人撰写。
2. 本项目基于上个项目 `CS205 C/C++ Programming -- Project 2 Simple Matrix Multiplication` 以及 `CS205 C/C++ Programming -- Project 3 Improved Matrix Multiplicatio`

`n`，所以忽略了对矩阵本身计算速度的探究，着重于探讨利用符号重载完成矩阵的运算以及对矩阵数据空间利用的优化。