CS205: Project 4: A Class to Describe a Matrix

杨彦卓 (南方科技大学，通识与学科基础部)

2024.05.16

摘　要

该文章介绍了矩阵的 c++ 代码实现过程，试图创建用共享指针来优化感兴趣区域的内存重复占用问题。

关键词：矩阵、内存管理

目录

1 前言

1.1 关于生成式人工智能的声明

对于代码中很多高相似部分，包括简单地重载操作符（如两个矩阵间的“+=”，“-=”、矩阵数乘等），代码由 ChatGPT 改写。但对于测试代码、撰写报告、图表设计等流程均不涉及任何生成式人工智能。

1.2 开发环境

1.2.1 IDE 配置

- **IDE**：CLion 2023.3.4
- **CMake Version**: 3.27
- **C++ 编译标准**：C++17

1.2.2 VSCode 测试搭载的 Ubuntu 版本

Ubuntu 20.04.2 LTS x86_64 with gcc version 9.4.0-1ubuntu1 20.04.2) 9.4.0

1.3 矩阵乘法库说明

对于矩阵乘法的优化已在前一个项目及其报告中完成，此次项目选择调用 OpenBLAS 库实现矩阵乘法。但对于其他矩阵操作，均不涉及 OpenBLAS 或任何其他外库。

OpenBLAS 参考网址超链接：https://www.openblas.net/

2 矩阵初始化

2.1 矩阵定义

我们使用模板来定义矩阵。其主要目的是为了实现数据类型的泛化和代码的重用。通过模板，矩阵类可以适用于不同的数据类型（如 int、float、double 等）。这样，同一份代码就能处理多种类型的数据，而无需为每种数据类型单独编写特定的类。

```
template<typename T>
```

Figure 1: 用该字段声明接下来的函数或类为模板类型

2.2 成员变量声明

该矩阵声明的所有成员变量均为 private。目的是为了实现数据封装和信息隐藏，有助于提高代码的安全性。成员变量的有关信息如下表：

变量数据类型	变量名	描述
size_t	rows	矩阵的行数
size_t	cols	矩阵的列数
std::shared_ptr<T[ ]>	data	指向矩阵数据的智能指针
bool	isROI	标识矩阵是否为 ROI
Matrix<T>*	parent	指向 ROI 源矩阵的指针

Table 1: Matrix 类的成员变量

2.3 通过数组初始化矩阵

2.3.1 代码实现

数组与矩阵在形式上有很多相似之处，所以我认为有必要建立一个构造器，根据数组来构造矩阵。在实现的过程中，我考虑了一维数组和二维数组两种情况并分别重载了构造函数。

```
template<typename T>
Matrix<T>::Matrix(size_t rows, size_t cols, const T *dataArray): rows(rows), cols(cols), isROI(false), parent(nullptr) {
    if (rows == 0 || cols == 0)
        throw std::invalid_argument("The dimension of the matrix can not be ZERO.");
    data = std::shared_ptr<T[]>(new T[rows * cols], std::default_delete<T[]>());
    std::memcpy( Dst: data.get(), Src: dataArray, Size: rows * cols * sizeof(T));
}
```

Figure 2: 用一维数组构造矩阵

```cpp
template <typename T>
Matrix<T>::Matrix(size_t rows, size_t cols, const T** dataArray)
        : rows(rows), cols(cols), isROI(false), parent(nullptr) {
    if (rows == 0 || cols == 0)
        throw std::invalid_argument("The dimension of the matrix can not be ZERO.");
    data = std::shared_ptr<T[]>(new T[rows * cols], std::default_delete<T[]>());
    for (size_t i = 0; i < rows; ++i) {
        std::memcpy( Dst: data.get() + i * cols,  Src: dataArray[i],  Size: cols * sizeof(T));
    }
}
```

Figure 3: 用二维数组构造矩阵

这种方法用循环结构赋值看似无懈可击，但我想到了另一个问题：对于特别是二维数组而言：在实际的赋值过程中，是否会出现这样一种情况：数组的尺寸与矩阵尺寸不匹配而导致代码运行异常？是否需要额外的检查来确保两者尺寸相匹配？由于数组在传参后退化为指针，所以很难在方法栈内直接获取数组的尺寸信息。但若在传参时加入两个参数来描述数组尺寸，这样的参数列表过长，便会显得代码十分臃肿。

经过测试后我发现代码并不会运行异常。于是我查阅了有关 memcpy 方法的说明与源码。

```cpp
void *__cdecl memcpy(void * __restrict__ _Dst,const void * __restrict__ _Src,size_t _MaxCount) __MINGW_ATTRIB_DEPRECATED_SEC_WARN;
```

Figure 4: memcpy 方法名以及参数列表

阅读后续源码可知对于 memcpy 原型函数：memcpy 只会按照 n 指定的字节数，从 src 给定的解引用位置开始进行内存复制，不会检查 src 和 dest 是否有效或者所谓的“匹配”。因此我们可以说，memcpy 假设调用者保证源和目标内存区域的有效性和大小。如果源数组足够大（至少包含 rows * cols 个元素），memcpy 将会成功复制数据而不会发生越界。如果源数组不足 rows*cols 个元素，但相邻内存区域没有被其他重要数据占据，memcpy 可能会读取这些无效但未被保护的内存区域。这种情况下，尽管数据不正确，但程序可能不会立刻崩溃。

了解到这些之后，不难发现这个用二维数组作为参数的构造器其实是多余的。假设我们直接用二维数组 arrayData 写入参数列表，那么在传参后它便会退化为一个指向第一个子数组的指针。因此，这样的做法不如在传参前就用 *arrayData 来指向第一个数组，并由此来调用“一维数组构造器”(因为 *arrayData 在传参后就退化为了指向第一个数组的第一个元素)。根据我们对于 memcpy 的了解，在后续的赋值过程中并不会出现其他问题。当然，源数组不足的问题仍然存在。但既然 memcpy 的撰写者都可以相信调用者会保证参数之间的有效性关系，那同理我定义的该构造器也能相信调用者会保证源数组与

矩阵尺寸之间的有效性关系。因为在绝大部分情况下，源数组不够时，memcpy 只会读取一些邻近的无效数据来填充，并非产生一些致命错误。

2.3.2 代码示例

```
int array[] = { [0]: 1,  [1]: 2,  [2]: 3,  [3]: 4,  [4]: 5,  [5]: 6};
int array_2D[2][3] = { [0]: { [0]: 1,  [1]: 2,  [2]: 3},
                       [1]: { [0]: 4,  [1]: 5,  [2]: 6}};
Matrix<int> matrix( rows: 2,  cols: 3,  dataArray: array);
Matrix<int> matrix_23( rows: 2,  cols: 3,  dataArray: *array_2D);
Matrix<int> matrix_32( rows: 3,  cols: 2,  dataArray: *array_2D);
cout << "Matrix constructed by 1D array: \n" << matrix << endl;
cout << "Matrix constructed by 2D array: \n" << matrix_23 << endl;
cout << "Matrix constructed by 2D array: \n" << matrix_32 << endl;
```

```
Matrix constructed by 1D array:
1 2 3
4 5 6

Matrix constructed by 2D array:
1 2 3
4 5 6

Matrix constructed by 2D array:
1 2
3 4
5 6
```

2.4 通过其他矩阵初始化原矩阵

2.4.1 代码实现

通常认为，对于一个矩阵的感兴趣区域 (region of interest, ROI) 应该与线性代数方面定义的子矩阵相似：我们应该关心如何从一个更大的源矩阵中，提取一部分在矩阵意义上连续的子矩阵 (而不是在计算机意义上内存连续的子内存空间)。如下图所示，黄色部分代表源矩阵，表示数据来源，绿色的部分表示 ROI 矩阵。为了避免内存硬拷贝，我们把源矩阵的数据从指定的 (startRows, startCols) 开始，通过共享指针将该位置后若干个连续的个数据共享给 ROI 矩阵。

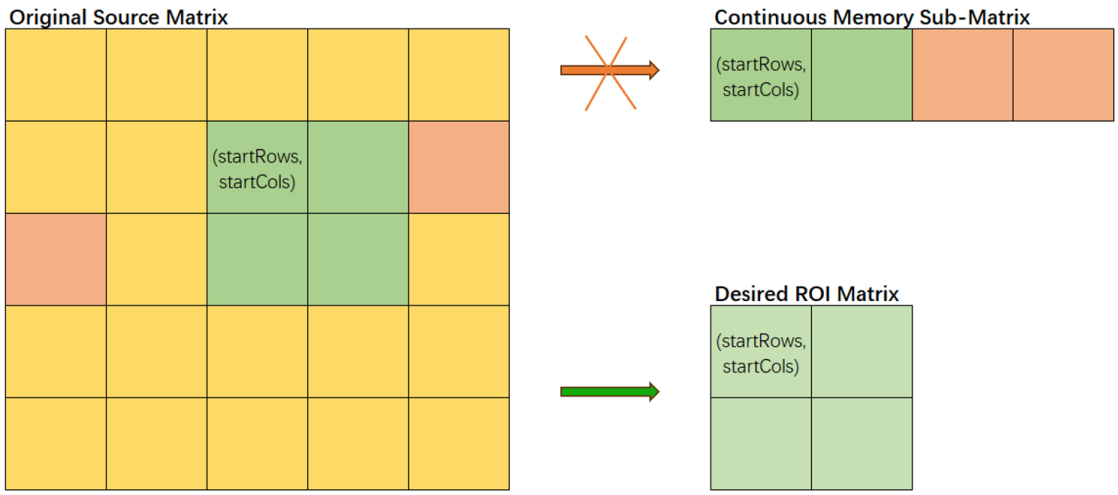

Figure 5: 子矩阵与 ROI 示意图

```
template<typename T>
Matrix<T>::Matrix(size_t rows, size_t cols, Matrix<T> &parentMat, size_t startRows, size_t startCols)
        : rows(rows), cols(cols), isROI(true), parent(&parentMat) {
    if (startRows + rows > parentMat.rows || startCols + cols > parentMat.cols)
        throw std::out_of_range("Sub-matrix dimensions are out of range.");
    data = std::shared_ptr<T[]>(parentMat.data, parentMat.data.get() + startRows * parentMat.cols + startCols);
}
```

Figure 6: ROI 矩阵代码实现

值得注意的是，在 ROI 矩阵内的数据仍然是源矩阵连续的数据，所以我们应该用一些偏移来修正对 ROI 矩阵的访问和赋值 (在后文有详细解释)。在实现 ROI 的过程中，我考虑过用其他的方法来实现 ROI 矩阵内的数据保持独立的连续——也就是说 ROI 内不存在多余的数据，只存储我们所关心的子矩阵数据 (根据示意图 5 来解释，便是 ROI 内只存放了 4 个绿色元素的数据)。但实际上 ROI 存放了从 start 位置开始之后的所有数据。

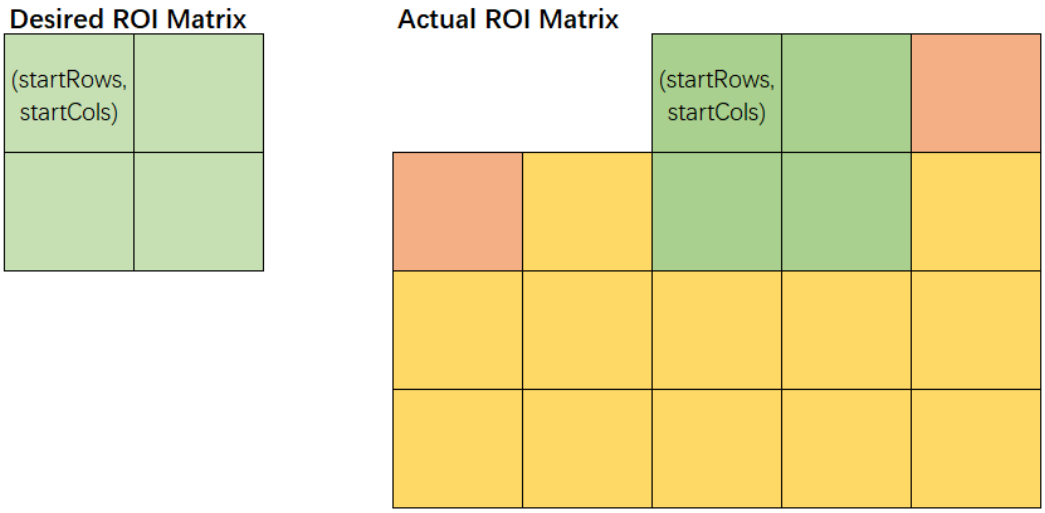

Figure 7: ROI 矩阵数据示意图

但以我有限的水平想到的很多方案都有着明显的缺陷：除了代码不够优雅以外，最重要的是很难避免额外的拷贝操作甚至硬拷贝。

共享指针设计的初衷本应该就是让我们通过实现多个指针共享同一块内存，其“智能”的标签就体现在利用计数的机制从而自动管理内存的生命周期。倘若再引用别的指针或内存空间或其他约束来辅助管理 ROI，那共享指针还有何必要存在？

所以我想到一个相对较优雅的方法——设计一个偏移量 (BIAS)，对访问或者赋值矩阵数据时进行修正，从而返回正确的数据。若试图超出理想 ROI 的管理范围，就拒绝此次访问。

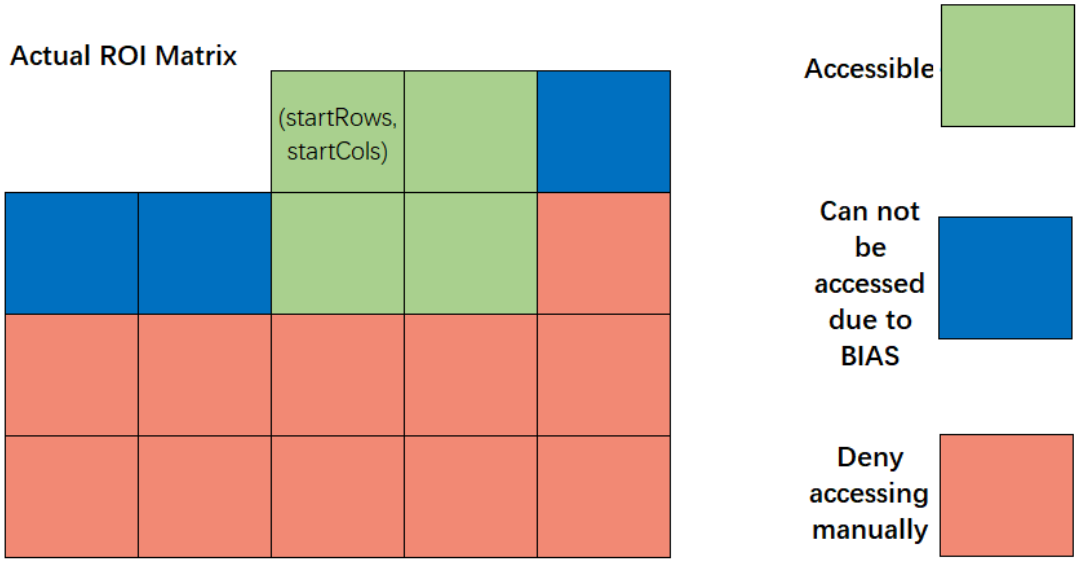

Figure 8: ROI 矩阵数据访问示意图

2.4.2 代码示例

```
int a[5][5] = {
        { 0: { 0: 1,  1: 2,  2: 3,  3: 4,  4: 5},
        { 1: { 0: 6,  1: 7,  2: 8,  3: 9,  4: 10},
        { 2: { 0: 11,  1: 12,  2: 13,  3: 14,  4: 15},
        { 3: { 0: 16,  1: 17,  2: 18,  3: 19,  4: 20},
        { 4: { 0: 21,  1: 22,  2: 23,  3: 24,  4: 25}
};
Matrix<int> matrix( rows: 5,  cols: 5,  dataArray: *a);
Matrix<int> ROI( rows: 2,  cols: 2,  &: matrix,  startRows: 1,  startCols: 2);
cout << "Original matrix: \n" << matrix << endl;
cout << "ROI : \n" << ROI << endl;
ROI.setValue( row: 0,  col: 0,  value: 999999);
cout << "Element in the 2nd.row, 3rd.column in the original matrix: " << matrix.getValue( row: 1,  col: 2) << endl;
printf( format: "%d", ROI( row: 2,  col: 2));  // Access Denied
```

```
Original matrix:
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

ROI :
8 9
13 14

Element in the 2nd.row, 3rd.column in the original matrix: 999999
terminate called after throwing an instance of 'std::out_of_range'
  what():  The desired index is out of range.
```

2.5 其他常用初始化或赋值矩阵方法

2.5.1 矩阵硬拷贝

```cpp
template <typename T>
Matrix<T> Matrix<T>::hardCopy() const {
    Matrix<T> copy(rows, cols);
    if (isROI && parent) {
        for (size_t i = 0; i < rows; ++i)
            for (size_t j = 0; j < cols; ++j)
                copy.data[i * cols + j] = (*this)(i, j);
    } else
        std::memcpy( Dst: copy.data.get(), Src: data.get(), Size: rows * cols * sizeof(T));
    return copy;
}
```

Figure 9: 硬拷贝代码实现

如果该矩阵本就是一个 ROI，也可以通过这种方法新建一个矩阵，重新硬拷贝一次源矩阵的数据。注意代码需要改写。因为在 ROI 内有效数据不是连续的。

```cpp
int array[5][5] = { [0]: { [0]: 1,  [1]: 2,  [2]: 3,  [3]: 4,  [4]: 5},
                    [1]: { [0]: 6,  [1]: 7,  [2]: 8,  [3]: 9,  [4]: 10},
                    [2]: { [0]: 11, [1]: 12, [2]: 13, [3]: 14, [4]: 15},
                    [3]: { [0]: 16, [1]: 17, [2]: 18, [3]: 19, [4]: 20},
                    [4]: { [0]: 21, [1]: 22, [2]: 23, [3]: 24, [4]: 25}};
Matrix<int> matrix( rows: 5, cols: 5, dataArray: *array);
Matrix<int> ROI( rows: 2, cols: 2, &: matrix, startRows: 1, startCols: 2);
Matrix<int> copy = ROI.hardCopy();
ROI( row: 0, col: 0) = 9999;
cout << "ROI won't affect copy: \n" << copy << endl;
```

```
ROI won't affect copy:
8 9
13 14
```

Figure 10: 硬拷贝代码示例

2.5.2 全零矩阵初始化

```cpp
template <typename T>
Matrix<T> Matrix<T>::zeros(size_t rows, size_t cols) {
    Matrix<T> zeroMatrix(rows, cols);
    for (size_t i = 0; i < rows * cols; ++i)
        zeroMatrix.data[i] = 0;
    return zeroMatrix;
}
```

Figure 11: 全零矩阵初始化代码实现

3 矩阵数据访问及数据赋值

3.1 常规方法

常规思路下，我们考虑写一个常规的 getValue 和 setValue 方法即可。注意将 getValue 用 const 关键字约束。

代码实现和示例过于简单，无需展示在纸面报告中，可移步至源码查看细节，略。

3.2 优化思路

为了方便访问矩阵中的指定索引的元素或对其进行赋值，我重写了“()”操作符使得访问和赋值更加方便。该操作符需要传入两个参数分别表示行列索引 (从 0 开始)，并返回矩阵数据中对应位置的数据。

在后续的实现过程中发现，我发现有必要在添加 const 关键字后再次重写，否则在某些 const 矩阵方法中可能无法正常调用该操作符来访问指定数据。在再次重写后，编译器可以确保数据在不被修改时得到保护，这样也能满足 C++ 中的 const-correctness 原则，增强代码的安全性和可维护性。

代码实现和示例略。

3.3 利弊分析

世间可比之万物，此优势即彼劣势，此劣势即彼优势。

- **常规封装方法**：相较于重写操作符，这些方法允许在数据访问和修改时进行额外的控制，如验证输入值或执行其他逻辑。最为重要的是，更准确的方法名更易于调试，方便我们更加具体地追踪调用了哪种方法。
- **重写操作符**：相较于常规方法，重写操作符显然大大提高了代码的可读性，极大程度上避免了代码冗长的问题。

3.4 对 ROI 中 bias 的特殊处理

3.4.1 代码实现

此处对于 ROI 中多余数据的产生不再赘述。我们以重载“()”操作符为例，讨论如何实现偏移。

```cpp
template<typename T>
T &Matrix<T>::operator()(size_t row, size_t col) {
    if (row >= rows || col >= cols)
        throw std::out_of_range("The desired index is out of range. ");
    if (isROI)
        return data[row * parent->cols + col];
    else
        return data[row * cols + col];
}
```

Figure 12: 访问或赋值矩阵数据代码实现

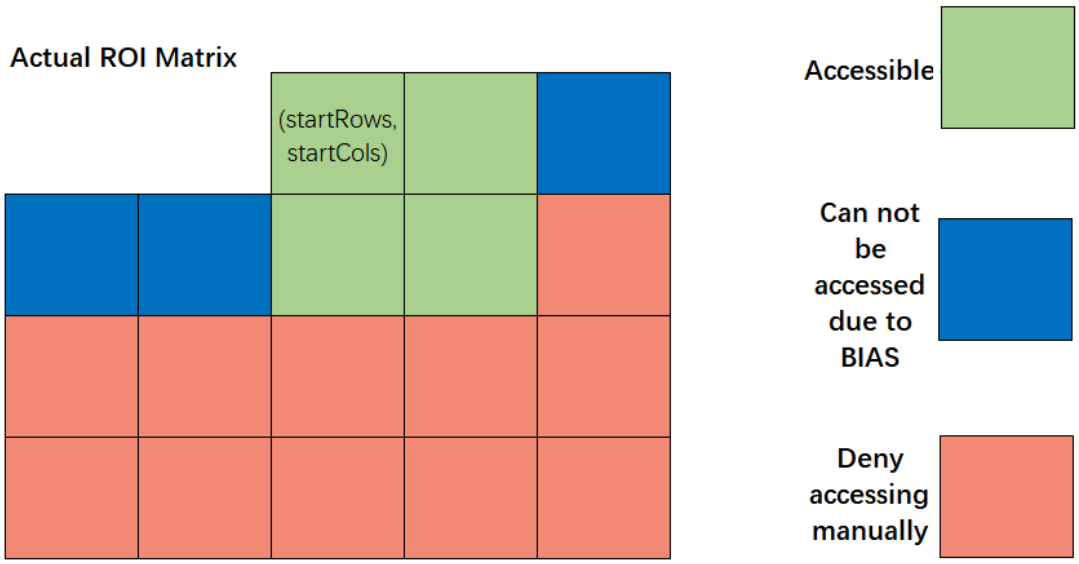

Figure 13: 回顾：ROI 矩阵数据访问示意图

首先判断访问的索引是否越界，这对于普通矩阵和 ROI 都是必须的。对于 ROI，这一步相对于筛选并过滤了红色元素。再判断该矩阵是否为 ROI，如果是的话需要用 row * parent->cols + col 修正所访问的数据位置。(注意：在初始化 ROI 时，其数据从源矩阵的第一个有效值开始！所以不需要考虑绿色元素之前的偏移！)

- row * parent->cols：跳转至指定行的起始位置。
- + col：在指定行内，跳转至指定列的位置。

经过修正以后，便可以像访问线性代数层面的子矩阵一样访问 ROI。

3.4.2 代码示例

```cpp
int a[5][5] = {
        [0]: { [0]: 1,  [1]: 2,  [2]: 3,  [3]: 4,  [4]: 5},
        [1]: { [0]: 6,  [1]: 7,  [2]: 8,  [3]: 9,  [4]: 10},
        [2]: { [0]: 11, [1]: 12, [2]: 13, [3]: 14, [4]: 15},
        [3]: { [0]: 16, [1]: 17, [2]: 18, [3]: 19, [4]: 20},
        [4]: { [0]: 21, [1]: 22, [2]: 23, [3]: 24, [4]: 25}
};
Matrix<int> matrix( rows: 5, cols: 5, dataArray: *a);
Matrix<int> ROI( rows: 3, cols: 3, &: matrix, startRows: 1, startCols: 2);
cout << "Get all elements in ROI through two loops: " << endl;
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j)
        cout << ROI( row: i, col: j) << " ";
    cout << endl;
}
cout << ROI( row: 3, col: 3) << endl;
```

```
Get all elements in ROI through two loops:
8 9 10
13 14 15
18 19 20
terminate called after throwing an instance of 'std::out_of_range'
  what():  The desired index is out of range.
```

Figure 14: 访问或赋值矩阵数据代码示例

4 矩阵功能

4.1 总述

以下表格列出了此处项目中实现的所有矩阵功能：

函数或方法及其参数列表	解释说明
Matrix(size_t rows, size_t cols)	构造函数，指定尺寸
Matrix(const Matrix &other)	拷贝构造函数，指定尺寸
Matrix(size_t rows, size_t cols, const T *dataArray)	通过给定的数据数组创建一个矩阵
Matrix(size_t rows, size_t cols, Matrix<T>& parentMat, size_t startRows, size_t startCols)	创建 ROI，从源矩阵的起始位置开始创建并指定特定尺寸
Matrix<T> hardCopy() const	创建并返回当前矩阵的一个硬拷贝
static Matrix<T> zeros(size_t rows, size_t cols)	静态函数，创建全零矩阵
Matrix &operator=(const Matrix &other)	赋值运算符重载，将一个矩阵赋值给另一个矩阵
T &operator()(size_t row, size_t col)	返回矩阵中指定位置的数据

函数或方法及其参数列表	解释说明
const T &operator()(size_t row, size_t col) const	返回矩阵中指定位置的数据的常量引用
T getValue(size_t row, size_t col) const	返回矩阵中指定位置的数据
void setValue(size_t row, size_t col, const T &value)	设置矩阵中指定位置的数据
Matrix operator+(const Matrix &other) const	重载加法运算符，实现两个矩阵的加法
Matrix operator-(const Matrix &other) const	重载减法运算符，实现两个矩阵的减法
Matrix operator*(const Matrix &other) const	重载乘法运算符，实现两个矩阵的乘法
Matrix operator+(const T &value) const	重载加法运算符，实现矩阵与常数的加法
Matrix operator-(const T &value) const	重载减法运算符，实现矩阵与常数的减法
Matrix operator*(const T &value) const	重载乘法运算符，实现矩阵与常数的乘法
Matrix operator/(const T &value) const	重载除法运算符，实现矩阵与常数的除法
bool operator==(const Matrix &other) const	重载相等运算符，比较两个矩阵的数据是否完全一样
bool operator!=(const Matrix &other) const	重载不等运算符，比较两个矩阵数地址是否不等
~Matrix()	析构函数，释放矩阵的资源。
Matrix<T> &operator+=(const Matrix &other)	重载加法赋值运算符，实现矩阵与矩阵的加法并赋值
Matrix<T> &operator-=(const Matrix &other)	重载减法赋值运算符，实现矩阵与矩阵的减法并赋值
Matrix<T> &operator*=(const Matrix &other)	重载乘法赋值运算符，实现矩阵与矩阵的乘法并赋值
Matrix<T> &operator+=(const T &value)	重载加法赋值运算符，实现矩阵与常数的加法并赋值
Matrix<T> &operator-=(const T &value)	重载减法赋值运算符，实现矩阵与常数的减法并赋值
Matrix<T> &operator*=(const T &value)	重载乘法赋值运算符，实现矩阵与常数的乘法并赋值

函数或方法及其参数列表	解释说明
Matrix<T> &operator/=(const T &value)	重载除法赋值运算符，实现矩阵与常数的除法并赋值
void print() const	打印矩阵信息：尺寸、数据类型以及数据
friend std::ostream &operator«<>(std::ostream &os, const Matrix<T> &matrix)	重载输出运算符，实现矩阵对象的输出
friend std::istream &operator»<>(std::istream &is, Matrix<T> &matrix)	重载输入运算符，实现从输入流中读取矩阵数据

4.2 常用或较为简单功能实现

很多矩阵功能其实较为简单，这里只列出一些代表性功能的代码：

4.2.1 赋值符号 =

此处的赋值是软拷贝。重载赋值符号中，赋值前应该避免自我赋值。

```
template<typename T>
Matrix<T>& Matrix<T>::operator=(const Matrix& other) {
    if (this == &other)
        return *this;
    rows = other.rows;
    cols = other.cols;
    isROI = other.isROI;
    parent = other.parent;
    data = other.data;
    return *this;
}
```

Figure 15: 重载赋值符号代码实现

4.2.2 相等运算符 == & 不等运算符!=

.

- **对于 == 的重载**：我们判断两个矩阵的数据在数值上是否完全一致，这更符合实际需求；
- **对于!= 的重载**：出于同样的考虑，我们判断两个矩阵的数据地址是否一致。

4.2.3 矩阵乘法

在开篇中已经提到，本次项目调用 OpenBLAS 实现矩阵乘法。

```
template<typename T>
Matrix<T> Matrix<T>::operator*(const Matrix& other) const {
    if (cols != other.rows) {
        throw std::invalid_argument("Matrix dimensions do not match for multiplication.");
    }

    Matrix<T> result(rows, other.cols);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                rows, other.cols, cols,
                1.0, data.get(), cols,
                other.data.get(), other.cols,
                0.0, result.data.get(), other.cols);
    return result;
}
```

Figure 16: 矩阵乘法代码实现

5 结语

5.1 学习心得

实践出真知。

很多想当然认为理解的知识点，在实际运用的过程中还是反反复复被查了好几次相关定义。知识就应该反复操练。Project 虽痛苦，但确实比做题得效果来的好得多、快得多。

5.2 感悟