Project 4

Jiawei Xiong SID:12011022

Introduction

The `Matrix` class is designed to accommodate various data types, providing flexibility for different applications. With essential data members such as `rows`, `cols`, and a pointer to matrix elements, the class ensures efficient storage and manipulation of 2D matrices.

Key requirements include proper memory management to avoid memory leaks and the implementation of operator overloading for intuitive matrix operations. Additionally, the class supports the concept of Region of Interest (ROI) to optimize memory usage by sharing memory between matrices.

By leveraging C++ features such as templates and operator overloading, the `Matrix` class offers ease of use and flexibility while maintaining memory safety. This introduction sets the stage for a comprehensive exploration of the class's implementation, highlighting its significance in advancing matrix operations within the C++ environment.

Class menbers

Private member

```
    private:
        size_t numRows, numCols;
        shared_ptr<T> elements;
        T* originalElements;
        bool isSubMatrix;
```

`numRows`, `numCols`: Using size_t variables to store the size of matrix.

`elements`: Using shared_ptr to dynamically allocate memory for matrix elements.

`originalElements`: A row pointer to the matrix elements, for ROI objects

`SubMatrix`: A boolean variable indicates whether the matrix uses ROI

Public member

Constructor:

primary constructor： This constructor is used to create a matrix object of specified size and allocate memory to store the matrix elements.

```
Matrix(size_t numRows, size_t numCols) : numRows(numRows), numCols(numCols),
isSubMatrix(false), originalElements(nullptr)
{elements = shared_ptr<T>(new T[numRows * numCols]);}
```

copy constructor： The copy constructor creates a new Matrix object by copying the dimensions and element data from another Matrix object.

```
Matrix(const Matrix& other) : numRows(other.numRows), numCols(other.numCols),
elements(other.elements) {}
```

ROI constructor :The ROI constructor creates a new Matrix object representing a region of interest within another Matrix object, sharing the same data memory but with a different size and offset.

```
 Matrix(size_t rowOffset, size_t colOffset, size_t numRows, size_t numCols, Matrix&
parent)
            : numRows(numRows), numCols(numCols), isSubMatrix(true),
originalElements(parent.elements.get()) {
            elements = shared_ptr<T>(parent.elements, originalElements + (rowOffset *
parent.numCols + colOffset));
        }
```

Destructor

The destructor releases the memory allocated for the matrix elements if the matrix is not a region of interest (ROI), ensuring memory cleanup and avoiding memory leaks.

```
~Matrix() {if (!isSubMatrix && elements) {
                elements.reset();}}
```

Index operator

```
        T& operator()(size_t row, size_t col) {
            return elements.get()[row * numCols + col];
        }

        const T& operator()(size_t row, size_t col) const {
            return elements.get()[row * numCols + col];
        }
```

These two operator() functions provide access to the elements of the matrix using the () operator, allowing both read and write access for non-const objects and read-only access for const objects.

Operator Overloads

“=” overload：

The copy assignment operator copies the contents of another Matrix object to the current object, ensuring proper memory management and handling of region of interest (ROI) objects to avoid unnecessary memory copying.

```
        Matrix& operator=(const Matrix& other) {
            if (this != &other) {
                numRows = other.numRows;
                numCols = other.numCols;
                isSubMatrix = other.isSubMatrix;
                originalElements = other.originalElements;
                if (!isSubMatrix) {
                    elements = shared_ptr<T>(new T[numRows * numCols]);
                    std::copy(other.elements.get(), other.elements.get() + numRows *
numCols, elements.get());
                } else {
                    elements = other.elements;
                }
            }
            return *this;
        }
```

“+” overload：

The operator+ overloads the addition operator to perform element-wise addition between two matrices, ensuring that their dimensions match and throwing an exception otherwise.

```
        Matrix operator+(const Matrix& other) const {
            if (numRows != other.numRows || numCols != other.numCols) {
                throw std::invalid_argument("矩阵维度必须匹配以进行加法操作。");
            }
            Matrix result(numRows, numCols);
            for (size_t i = 0; i < numRows * numCols; i++) {
                result.elements.get()[i] = elements.get()[i] + other.elements.get()[i];
            }
            return result;
        }
```

“-” overload：

The operator- overloads the subtraction operator to perform element-wise subtraction between two matrices, ensuring that their dimensions match and throwing an exception otherwise.

```
 Matrix operator-(const Matrix& other) const {
            if (numRows != other.numRows || numCols != other.numCols) {
                throw std::invalid_argument("矩阵维度必须匹配以进行减法操作。");
            }
            Matrix result(numRows, numCols);
            for (size_t i = 0; i < numRows * numCols; ++i) {
                result.elements.get()[i] = this->elements.get()[i] - other.elements.get()
[i];
            }
            return result;
        }
```

“*” overload：

The operator* overloads the multiplication operator to perform element-wise multiplication between two matrices, ensuring that their dimensions match and throwing an exception otherwise.

```
Matrix operator*(const Matrix& other) const {
            if (numRows != other.numRows || numCols != other.numCols) {
                throw std::invalid_argument("矩阵维度必须匹配以进行元素逐一相乘操作。");
            }
            Matrix result(numRows, numCols);
            for (size_t i = 0; i < numRows * numCols; ++i) {
                result.elements.get()[i] = this->elements.get()[i] * other.elements.get()
[i];
            }
            return result;
        }
```

“dot” overload：

The dot function performs matrix multiplication between two matrices, resulting in a new matrix with dimensions appropriate for the multiplication, ensuring that the number of columns in the first matrix matches the number of rows in the second matrix, and throwing an exception otherwise.

```
        Matrix dot(const Matrix& other) const {
            if (numCols != other.numRows) {
                throw std::invalid_argument("矩阵维度必须匹配以进行矩阵乘法操作。");
            }
            Matrix result(numRows, other.numCols);
            for (size_t i = 0; i < numRows; i++) {
                for (size_t j = 0; j < other.numCols; j++) {
                    result(i, j) = 0;
                    for (size_t k = 0; k < numCols; k++) {
                        result(i, j) += (*this)(i, k) * other(k, j);
                    }
                }
            }
            return result;
        }
```

“==” overload：

The operator== overloads the equality operator to compare two matrices for equality, returning true if they have the same dimensions and contain the same elements, and false otherwise.

```
bool operator==(const Matrix& other) const {
            if (numRows != other.numRows || numCols != other.numCols) return false;
            for (size_t i = 0; i < numRows * numCols; i++) {
                if (elements.get()[i] != other.elements.get()[i]) return false;
            }
            return true;
        }
        }
```

Print Function:

The print function prints the elements of the matrix to the standard output, row by row, separated by spaces and with each row on a new line.

```
        void print() const {
            for (size_t i = 0; i < numRows; i++) {
                for (size_t j = 0; j < numCols; j++) {
                    std::cout << (*this)(i, j) << " ";
                }
                std::cout << std::endl;}}};
```

Main Function

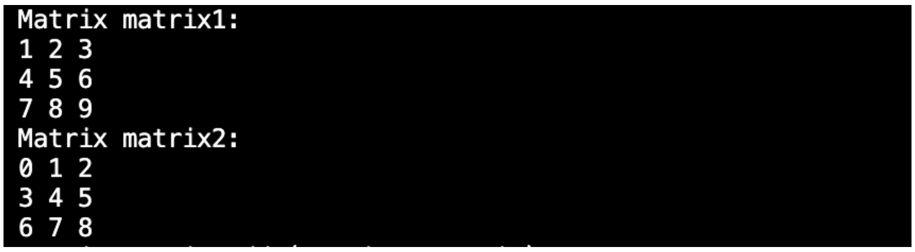
The `main` function initializes two 3x3 matrices, `matrix1` and `matrix2`, and populates them with consecutive integer values.

```
    Matrix<int> matrix1(3, 3);

    int count = 1;
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 3; j++) {
            matrix1(i, j) = count++;
        }
    }
    cout << "Matrix matrix1:\n";
    matrix1.print();
    Matrix<int> matrix2(3, 3);

    count =0;
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 3; j++) {
            matrix2(i, j) = count++;
        }
    }
    cout << "Matrix matrix2:\n";
    matrix2.print();
```

Then, it prints the contents of `matrix1` and `matrix2` to the console.



```
Matrix matrix1:
1 2 3
4 5 6
7 8 9
Matrix matrix2:
0 1 2
3 4 5
6 7 8
```

Next, it performs several matrix operations:

1. Addition: It adds `matrix1` and `matrix2` using the `+` operator, storing the result in `matrix_add`, and prints the result.

    ```
    Matrix<int> matrix_add = matrix1 + matrix2;
    cout << "Matrix matrix_add (matrix + matrix):\n";
    matrix_add.print();
    ```

2. Element-wise multiplication: It multiplies `matrix1` and `matrix2` element-wise using the `*` operator, storing the result in `matrix_mul`, and prints the result.

    ```
    Matrix<int> matrix_mul= matrix1 * matrix2;
    cout << "Matrix matrix_mul (matrix * matrix):\n";
    matrix_mul.print();
    ```

3. Matrix multiplication: It performs matrix multiplication between `matrix1` and `matrix2` using the `dot` method, storing the result in `matrix_dot`, and prints the result.

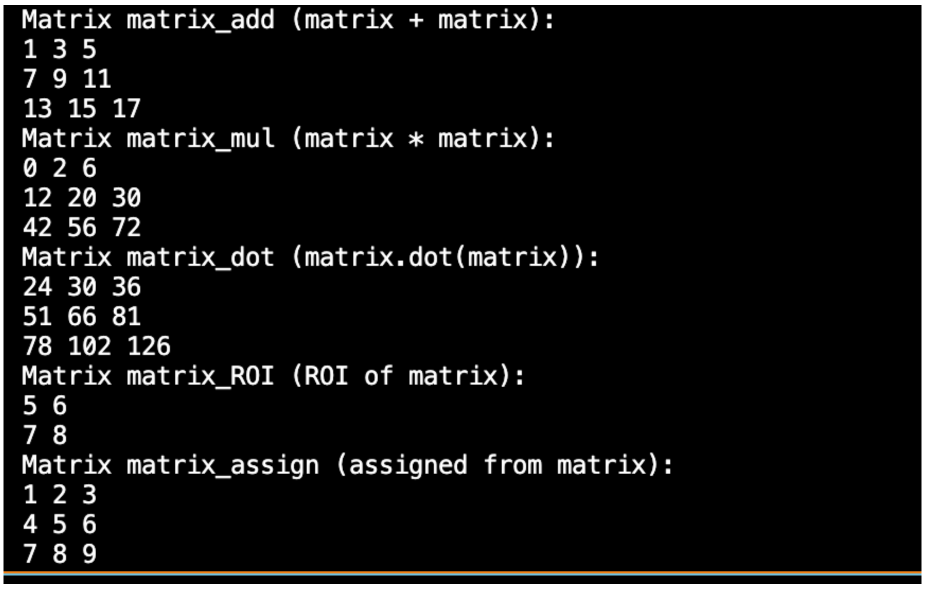
    ```
    Matrix<int> matrix_dot = matrix1.dot(matrix2);
    cout << "Matrix matrix_dot (matrix.dot(matrix)):\n";
    matrix_dot.print();
    ```

4. Region of Interest (ROI) test: It creates a region of interest (`matrix_ROI`) within `matrix1`, starting from row 1 and column 1 with a size of 2x2, and prints the ROI matrix.

    ```
    Matrix<int> matrix_ROI(1, 1, 2, 2, matrix1);
    cout << "Matrix matrix_ROI (ROI of matrix):\n";
    matrix_ROI.print();
    ```

5. Matrix assignment: It assigns `matrix1` to `matrix_assign` using the assignment operator, and prints the assigned matrix.

```
Matrix<int> matrix_assign = matrix1;
cout << "Matrix matrix_assign (assigned from matrix):\n";
matrix_assign.print();
```



```
Matrix matrix_add (matrix + matrix):
1 3 5
7 9 11
13 15 17
Matrix matrix_mul (matrix * matrix):
0 2 6
12 20 30
42 56 72
Matrix matrix_dot (matrix.dot(matrix)):
24 30 36
51 66 81
78 102 126
Matrix matrix_ROI (ROI of matrix):
5 6
7 8
Matrix matrix_assign (assigned from matrix):
1 2 3
4 5 6
7 8 9
```