Project 4: Matrix Class

SID	NAME
12212108	Bai Junyan

Introduction

In project 3, we have improved the performance of matrix multiplication using matrix blocking technique. In this project, we are going a stride further: implementing a matrix class with multiple utility functions. We aim to implement a matrix class, learned from `cv::Mat` of `OpenCV`, which satisfies several requirements:

- Concise and elegant design
- Safe memory management
- Highly efficient computation
- Convenient and user-friendly API

And there are also some tasks needed to be done:

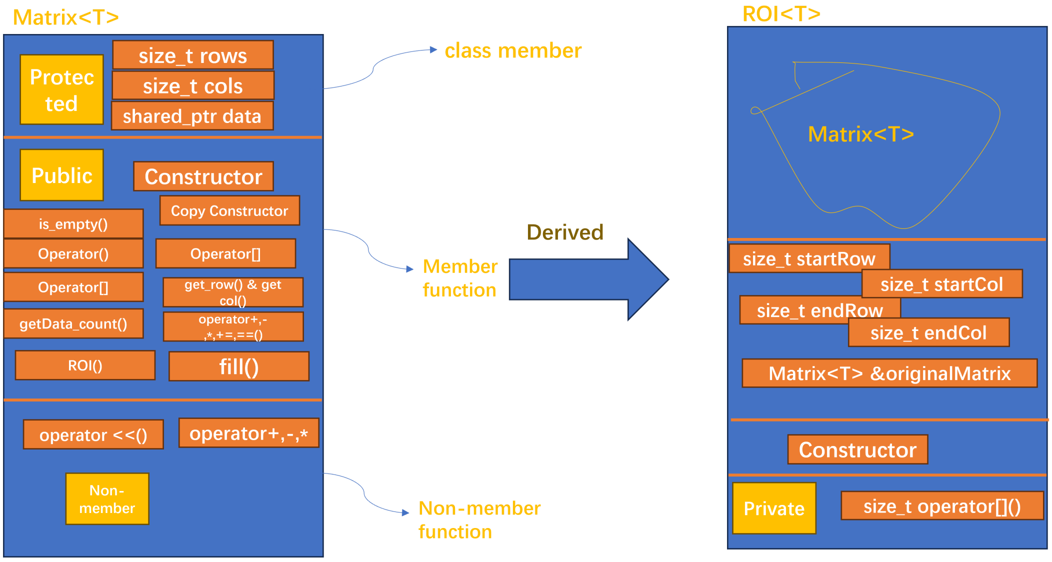
- Different datatypes of matrix should be supported.
- Memory hard copy should be avoided.
- Operation loading is required.
- ROI(Region of Interest) should be implemented.

Implementation

The structure of this project is as follow:

```
.
├─── README.md
├─── a.out
├─── main.cpp
├─── matrix.cpp
├─── matrix.h
├─── test.cpp
└─── try.cpp
```

The definition of my matrix class is in `matrix.h`, and the implementation is in `matrix.cpp`, test codes are in `main.cpp` and `try.cpp`. Let's take a closer look at the matrix class, as the picture below shows:



Different Data Types supported

There are mainly two classes: `Matrix` and `ROI`, which are both template class supporting multiple datatypes, where `ROI` is the derived class of its base class `Matrix`.

When using generic `T` to represent multiple datatypes, there are many "traps", which cause a few bugs:

Separate definition and implementation without `template <typename T>` before every implemented function and missing `Matrix<T>::Matrix` for member function implementation will cause **compiling error**. And not including `matrix.cpp` when creating object of template class will cause **link error**. Solutions can be found in this blog: https://www.codeproject.com/Articles/48575/How-to-define-a-template-class-in-a-h-file-and-imp.

Let's see the test result: Creating objects of different types like `double`, `int`, `char`, `string` shows:

```
Matrix<double> mat1(3, 3);
```

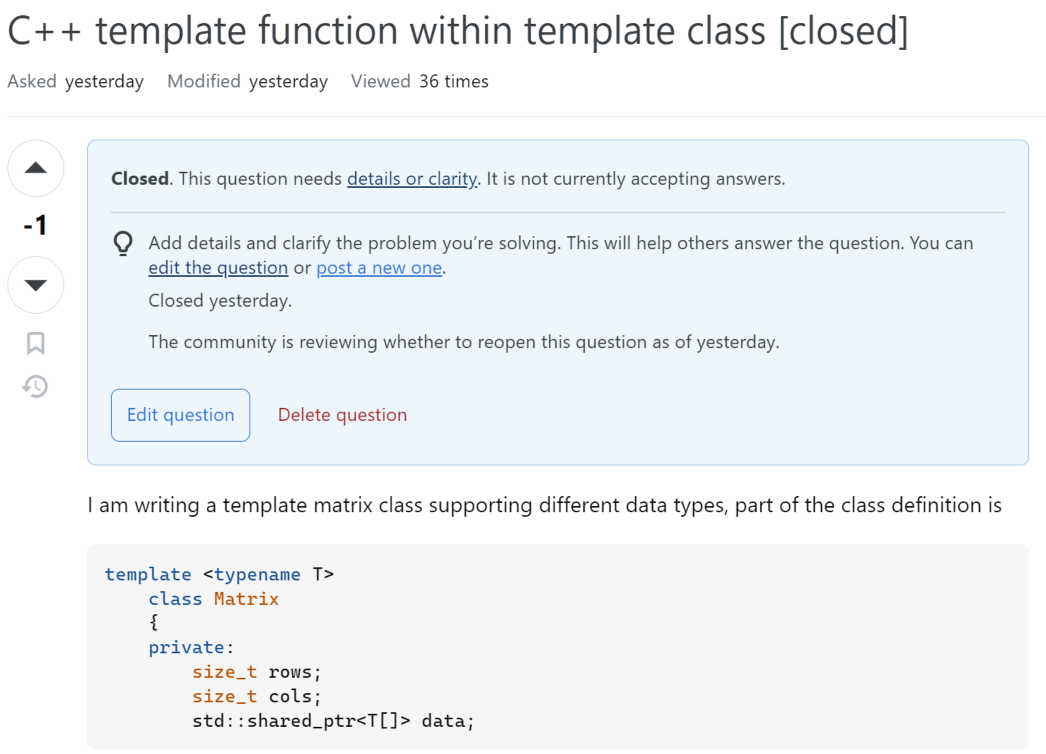
```
    Matrix<double> mat2(3, 3);

    mat1.fill(1.0); // Fill mat1 with 1.0
    mat2.fill(2.0); // Fill mat2 with 2.0

    std::cout << "Double Matrix 1:" << std::endl;

    Matrix<int> mat1(3, 3);
    Matrix<int> mat2(3, 3);

    mat1.fill(1); // Fill mat1 with 1
    mat2.fill(2); // Fill mat2 with 2

    std::cout << "Int Matrix 1:" << std::endl;
    std::cout << mat1;

    Matrix<std::string> mat1(2, 2);
    mat1(0, 0) = "Hello";
    mat1(0, 1) = "World";
    mat1(1, 0) = "Foo";
    mat1(1, 1) = "Bar";

    std::cout << "String Matrix 1:" << std::endl;

Double Matrix 1:
1 1 1
1 1 1
1 1 1

Int Matrix 1:
1 1 1
1 1 1
1 1 1

String Matrix 1:
Hello World
Foo Bar
Element at (0, 0) in mat1: Hello
Element at (1, 1) in mat1: Bar
```

,

Memory Hard Copy Avoided

Memory hard copy results in low efficiency during implementation. In this project, smart pointer is used to avoid it: each time the matrix is assigned to another matrix, smart pointer is shared among these matrices. The number of matrices sharing the same data memory can be checked by calling the method `int getData_count() const;`

Why don't I use `std::vector<T>`? Interestingly, when I post a question on stack overflow, experienced programmers suggest me using it instead:



C++ template function within template class [closed]

Asked yesterday Modified yesterday Viewed 36 times

Closed. This question needs details or clarity. It is not currently accepting answers.

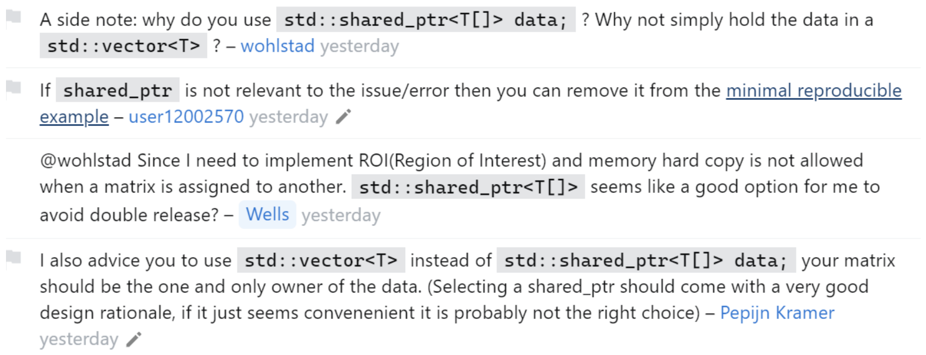
Add details and clarify the problem you're solving. This will help others answer the question. You can edit the question or post a new one.

Closed yesterday.

The community is reviewing whether to reopen this question as of yesterday.

I am writing a template matrix class supporting different data types, part of the class definition is

```
template <typename T>
    class Matrix
    {
    private:
        size_t rows;
        size_t cols;
        std::shared_ptr<T[]> data;
```



A side note: why do you use `std::shared_ptr<T[]> data;` ? Why not simply hold the data in a `std::vector<T>` ? – wohlstad yesterday

If `shared_ptr` is not relevant to the issue/error then you can remove it from the minimal reproducible example – user12002570 yesterday

@wohlstad Since I need to implement ROI(Region of Interest) and memory hard copy is not allowed when a matrix is assigned to another. `std::shared_ptr<T[]>` seems like a good option for me to avoid double release? – Wells yesterday

I also advice you to use `std::vector<T>` instead of `std::shared_ptr<T[]> data;` your matrix should be the one and only owner of the data. (Selecting a shared_ptr should come with a very good design rationale, if it just seems convenenient it is probably not the right choice) – Pepijn Kramer yesterday

I use smart pointer out of several reasons:

- convenience when avoiding memory issue.
- Soft copy and ROI should be implemented.
- I am trying to learn to use the rather new smart pointer.

I would try `std::vector<T>` afterwards.

Testing the copy method by writing:

```
WellsMatrixLib::Matrix<double> a(5, 5);
    a.fill(5);
    WellsMatrixLib::Matrix<double> b = a;
    WellsMatrixLib::Matrix<double> e(a);
    a.fill(5);
    std::cout << a.getData_count() << std::endl;
    std::cout << a << std::endl;
```

The result is:

```
3
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
```

It can be shown that data is indeed shared among the matrices.

Operation overloaded supported

As the picture above showed, basic operations on matrix are supported by operator overloading. They are defined like this:

```
        Matrix<T> operator+(const Matrix<T> &) const;
        Matrix<T> operator-(const Matrix<T> &) const;
        Matrix<T> operator*(const Matrix<T> &) const;
        Matrix<T> operator=(const Matrix<T> &); // Shallow copy
        Matrix<T> operator+=(const Matrix<T> &);
        // Matrix-Scalar Arithmetic
        Matrix<T> operator+(T) const;
        Matrix<T> operator-(T) const;
        Matrix<T> operator*(T);

        bool operator==(const Matrix<T> &) const;

        template <typename U>
        friend std::ostream &operator<<(std::ostream &, const Matrix<U> &);
        template <typename U>
        friend Matrix<U> operator+(T, const Matrix<T> &);
```

member and friend functions are both used to support both matrix matrix operation and matrix scalar operation. To improve efficiency, some techniques are used. For example, for addition, pointer is used to maintain cache locality:

```
size_t length = this->cols * this->rows;

        for (size_t i = 0; i < length; ++i)
        {

            result[i] = (*this)[i] + other[i];
        }

        return result;
```

And for matrix multiplication, `OpenBLAS` library is imported if compiled with static or dynamic `OpenBlas` library binding as wells as the defined macro `WITH_OpenBLAS` :

```
#ifndef WITH_OpenBLAS
        for (size_t i = 0; i < this->rows; i++)
        {
            for (size_t k = 0; k < this->cols; k++)
            {
                for (size_t j = 0; j < other.cols; j++)
                {
                    result(i, j) += this->data[i * this->cols + k] * other(k, j);
                }
            }
        }
#else
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, rows, other.cols, cols, 1.0, data.get(), cols, other.data.get(), other.cols, 0.0, result.data.get(), other.cols);
#endif
        return result;
```

Tests are shown below:

```
WellsMatrixLib::Matrix<double> a(5, 5);
    WellsMatrixLib::Matrix<double> b(5, 5);
    WellsMatrixLib::Matrix<double> e(a);
    a.fill(5);
    b.fill(6);
    std::cout << a(1, 1) << std::endl;
    WellsMatrixLib::Matrix<double> c = 5.0 + a;
    std::cout << c << std::endl;
    c += a;
    WellsMatrixLib::ROI<double> f = e.roi(2, 3, 2, 3);
    f(1, 1) = 111;
    WellsMatrixLib::Matrix<double> h(2, 2);
    h.fill(0, 1, 0, 1, 6);
    WellsMatrixLib::Matrix<double> s = f * h;
    std::cout << a.get_row() << std::endl;
    std::cout << a.get_col() << std::endl;
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << c << std::endl;
    std::cout << e << std::endl;
    std::cout << (c == a) << std::endl;
    // std::cout << d << std::endl;
    std::cout << e.getData_count() << std::endl;
    std::cout << f(1, 1) << std::endl
              << std::endl;
    std::cout << f.get_row() << std::endl;
    std::cout << f.getData_count() << std::endl;
    std::cout << h << std::endl;
    std::cout << s << std::endl;


wells@LegionWells ~/C++/Project4 (master*?) $ ./a.out
5
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10

5
5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 111 5
5 5 5 5 5

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

15 15 15 15 15
15 15 15 15 15
```

```
15 15 15 15 15
15 15 15 15 15
15 15 15 15 15

5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 111 5
5 5 5 5 5

0
2
111

2
2
6 6
6 6

60 60
696 696
```

ROI Implemented

ROI stands for "Region of Interest", is a term commonly used in the fields of image processing, computer vision, and related areas. It refers to a specific portion of an image that is of particular importance or significance for a given application or analysis. The ROI can be of any shape, such as rectangular, circular, polygonal, or irregular, depending on the requirements of the application.

A "middle layer" class `ROI` is implemented, which is the child of `Matrix`. The data member `startRow`, `startCol`, `endRow` and `endCol` are used to crop the designated field of the original matrix `&originalMatrix`. The `()` operator is overloaded in `ROI` as:

```
template <typename T>
    T &ROI<T>::operator()(size_t row, size_t col)
    {
        if (row < 0 || row + startRow >= originalMatrix.get_row() || col < 0 || col + startCol >= originalMatrix.get_col())
        {
            throw std::out_of_range("Index out of ROI range!");
        }
        return originalMatrix(startRow + row, startCol + col);
    }
```

When calling the method `roi` for matrix object, a ROI type matrix is generated which can be used to do further operations, which all derived from its base class(some need some slight modifications since the `data` pointer is not actually used for `ROI`).

```
template <typename T>
    ROI<T> Matrix<T>::Matrix::roi(size_t start_row, size_t end_row, size_t start_col, size_t end_col)
    {
        if (start_row >= rows || end_row >= rows ||
            start_col >= cols || end_col >= cols)
        {
            throw std::out_of_range("ROI index out of range!");
        }
        return ROI<T>(*this, start_row, end_row, start_col, end_col);
    }
```

The test results can be seen in part **Operation loaded**.

Other useful implementations

`fill` function is overloaded, where `std::fill` is used in the function.

```
size_t length = (end_row - start_row + 1) * (end_col - start_col + 1);
        auto start_pos = data.get() + (start_row * cols) + start_col;
        std::fill(start_pos, start_pos + length, num);
        return true;
```

operator "[]" is also implemented for internal use, so it is private:

```
size_t c = index % this->cols;
        size_t r = index / this->cols;
        return (*this)(r, c);
```

Conclusions

Implementing matrix class is a good practice of object-oriented coding and learning about c++ class. Due to piled projects, the time is rather urgent for me to complete this project compared to the 3 projects before. Still, lots have been learned while debugging and discussing with experienced programmers. C++ is indeed fascinating and elegant!