CS205-Project-5-A-Class-for-Matrices

CS205 2022-Fall Project 5 - A Class for Matrices

12110804 方嘉玮

Part 0 - Link

IskXCr/CS205-Project-5-A-Class-for-Matrices: CS205 2022-Fall Project 5 - A Class for Matrices (github.com)

Part 1 - List of Satisfied Requirements

We aim to provide a compact class for matrices that is both **simple**, **straightforward to use**, and **supports arbitrary data types (that can be stored)**.

- ***N-dimensions***, where `N` is a parameter that can vary from `0` to many, without specialized code for every dimension.
    - By supporting N-dimension matrix, matrices with multiple channels are automatically supported.
- Arbitrary types to store.
- Basic mathematical operations support, including per-element `+`, `-`, `*`, `/`, `%` for each element (if the type `T` supports) supported by built-in `apply()` function, and Matrix operations, including `+` and `*`.
- Subscripting support with both **Fortran-style** (`m(1,2,3)`), and **C-style** (`m[1][2][3]`).
- Move assignment and move constructor to ensure efficient passing of `Matrix` results and to eliminate expensive temporaries. (Use of `std::vector` and `std::array`, etc., with default constructors generated by the compiler.)
- **Region of Interest, or ROI**, supported by `Matrix_ref`s and `Matrix_slice`.
- Absence of resource leaks in the form of basic guarantee. (Using `std::vector` and other methods)
- Support constructor with an `initializer_list` argument that ease the initialization process.

Defects:

- All operations are written in a single `.hpp` file, because time is limited, and transplanting function with `template`s is hard. If time is enough, they can all be transferred to another `.cpp` file.
    - I believe the idea behind this is more important. That is, the usage of recursive templates and standard library routines, memory management, etc.
- Class template `Matrix_base` is not abstract enough, leading to copies of codes (between `Matrix_Ref` and `Matrix`, and also those specializations when `N=1` and `N=0` (They are used to speed up the performance, but they also lead to lots of duplicated codes). The usage of `iterator` and `const iterator` should be extracted from `Matrix_ref` and `Matrix` instead of being instantiated from two separate classes.

    - Haven't considered abstraction to be that difficult, but it turned out to be really difficult.
- Only basic structure is presented, for example per-element `apply` which applies a lambda function.
    - Through the use of subscripting, matrix multiplication and other operation can be easily implemented.
    - The correctness can only be guaranteed on the existing code. Further investigation is needed to complement (or refactor) this project.

Part 2 - Source Code & Some Tests

Matrix Structure

The structure is as follows:

```
// Base class: Matrix_base
template <typename T, size_t N>
class Matrix_base
{
    protected:
    Matrix_slice<N> desc; // the descriptor of this matrix
    // ...
}

// Reference class: Matrix_ref
template <typename T, size_t N>
class Matrix_ref : public Matrix_base<T, N>
{
    private:
    T *ptr; // The pointer to original elements
    // ...
}

// Basic class: Matrix
template <typename T, size_t N>
class Matrix : public Matrix_base<T, N>
{
    protected:
    std::vector<T> elems; // storing the elements of the matrix
    // ...
}
```

By using `std::vector` we get the benefit of avoiding manually managing memory resources, with efficient default move constructor, assignments... provided by the compiler. We explicitly state that we want each of the default constructor.

```
// default constructors
Matrix() = default;
Matrix(Matrix &&) = default; // Move constructor
Matrix &operator=(Matrix &&) = default;
Matrix(Matrix const &) = default;
Matrix &operator=(Matrix const &) = default;
~Matrix() = default;
```

Dimensional-Initialization with Each Element Default Initialized

```
// In class Matrix_slice:
template <typename... Exts>
Matrix_slice(Exts... exts)
{
    static_assert(Matrix_impl::Requesting_element<Exts...>(), "Matrix_slice:
invalid dimension argument.");
    extents = {size_t(exts)...};
    recalc_size();
}

// In class Matrix:
template <typename... Exts>
explicit Matrix(Exts... exts)
{
    // std::cerr << "Matrix(Exts constructor) Entered." << std::endl;
    Matrix_base<T, N>::desc = Matrix_slice<N>{exts...};
    // std::cerr << "Matrix(Exts constructor): Slice created." << std::endl;
    Matrix_base<T, N>::desc.init_full_dim();
    // std::cerr << "Matrix(Exts constructor): Dimension initialized. The target
size is " << Matrix_base<T, N>::desc.size << std::endl;
    elems.resize(Matrix_base<T, N>::desc.size);
}
```

Specializations are also contained in the source code, for dimension `N=0` and `N=1`. From this constructor you are able to do the following:

```
Matrix<int, 2> mat5(2, 2);
cout << "The row is " << mat5.rows() << endl;
cout << "The cols is " << mat5.columns() << endl;
mat5[1][1] = 5;
cout << "Brace-initialized mat5 with element [1][1] = " << mat5[1][1] << endl;
```

Higher dimension is also allowed.

```
Matrix<int, 3> mat6(3, 7, 3);
mat6[2][6][2] = 2333;
cout << "Brace-initialized mat6 with element [3][7][3] = " << mat6[2][6][2] <<
endl;

Matrix<int, 1> mat7(2);
mat7[1] = 4;
cout << "Brace-initialized mat7 with element [1] = " << mat7[1] << endl;

Matrix<int, 0> mat8;
mat8 = 6444;
cout << mat8() << endl;
```

If not particularly mentioned, all the methods below support *N-dimensional operations*.

List Initialization

This part is used to extract the dimensions from a `std::initializer_list` and load the elements into the `Matrix`.

```
namespace Matrix_impl
{
    /**
         * @brief For initializing a matrix using list initializer
         *
         * @tparam T
         * @tparam N
         */
    template <typename T, size_t N>
    struct Matrix_init
    {
        using type = std::initializer_list<typename Matrix_init<T, N -
1>::type>;
    };

    template <typename T>
    struct Matrix_init<T, 1>
    {
        using type = std::initializer_list<T>;
    };

    template <typename T>
    struct Matrix_init<T, 0>; // Undefined on purpose

    /**
         * @brief Check if the initializer_list is well formed.
         *
         * @tparam List
         * @param list
         * @return true
         * @return false
         */
    template <typename List>
```

```
    bool check_non_jagged(const List &list)
    {
        auto i = list.begin();
        for (auto j = i + 1; j != list.end(); ++j)
            if (i->size() != j->size())
                return false;
        return true;
    }

    template <size_t N, typename l, typename List>
    Enable_if<(N == 1), void> add_extents(l &first, const List &list)
    {
        *first++ = list.size();
    }

    /**
         * @brief Recursively add extents to the array, the outmost extent
first.
         *
         * @tparam N
         * @tparam l
         * @tparam List
         */
    template <size_t N, typename l, typename List>
    Enable_if<(N > 1), void> add_extents(l &first, const List &list)
    {
        assert(check_non_jagged(list));
        *first = list.size();
        add_extents<N - 1>(++first, *list.begin());
    }

    /**
         * @brief Return an `array` of extents from an `initializer_list`
         *
         * @tparam N
         * @tparam List
         * @param list
         * @return std::array<size_t, N>
         */
    template <size_t N, typename List>
    std::array<size_t, N> derive_extents(const List &list)
    {
        std::array<size_t, N> a;
        auto f = a.begin();
        add_extents<N>(f, list);
        return a;
    }

    /**
         * @brief Insert a sequence into the vector when dimension is 1.
         *
         * @tparam T
         * @tparam Vec
         * @param first
         * @param last
```

```
         * @param vec
         */
    template <typename T, typename Vec>
    void add_list(const T *first, const T *last, Vec &vec)
    {
        vec.insert(vec.end(), first, last);
    }

    template <typename T, typename Vec>
    void add_list(const std::initializer_list<T> *first, const
std::initializer_list<T> *last, Vec &vec)
    {
        for (; first != last; ++first)
            add_list(first->begin(), first->end(), vec);
    }


    template <typename T, typename Vec>
    void insert_flat(std::initializer_list<T> list, Vec &vec)
    {
        add_list(list.begin(), list.end(), vec);
    }

    // ...
};

// In Matrix<T, N> ...
// Matrix<T, 0> is specialized, and thus the corresponding initializer is
different.

/**
 * @brief Construct a new Matrix object using list-initialized constructor.
 *
 * @param init
 * @note
 * Rules:
 * If either a default constructor or an initializer-list constructor could be
invoked, prefer the default constructor.
 * If both an initializer-list constructor and an "ordinary constructor" could
be invoked, prefer the initializer-list constructor.
*/
template <typename T, size_t N>
class Matrix : public Matrix_base<T, N>
{
protected:
    std::vector<T> elems; // storing the elements of the matrix

public:
    // ...
    Matrix(Matrix_initializer<T, N> init)
    {
        Matrix_base<T, N>::desc.extents = Matrix_impl::derive_extents<N>(init);
        Matrix_base<T, N>::desc.recalc_size();
        Matrix_base<T, N>::desc.init_full_dim();
        elems.reserve(Matrix_base<T, N>::desc.size);
```

```
        Matrix_impl::insert_flat(init, elems);
        assert((elems.size() == Matrix_base<T, N>::desc.size));
    }
    // ...
}
```

Usage:

```
Matrix<double, 2> mat = {
        {2, 3, 3, 3},
        {6, 7, 8, 9},
        {2, 1, 5, 7}};
// or similarly
Matrix<double, 3> mat2 = {
    {{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}}};
```

If you get the wrong dimension in the list initializer, you will be warned (if `<cassert>` is included without `-DNDEBUG`):

```
Matrix<double, 3> mat2 = {
    {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
    {{10, 11, 12}, {13, 14}, {16, 17, 18}},
    {{19, 20, 21}, {22, 23, 24}, {25, 26, 27}}};
```

Output:

```
Assertion `(elems.size() == Matrix_base<T, N>::desc.size)' failed.
Aborted
```

Utilizing move constructors and assignments for intermediate calculation results

```
// Ok, compilers are supposed to handle this
Matrix(Matrix &&) = default;
Matrix &operator=(Matrix &&) = default;
```

- It is not hard copy right now, instead handled with care (possibly by `std::move`)
- In the case, that the only member in a `Matrix` is a `std::vector` and another class that uses default constructor and assignment operators, compiler should do better (at least for those default operations, to my understanding).
- For access and referencing the same matrix, we use `Matrix_slice` and `Matrix_ref`, which is a referential `Matrix` that allows both `read` and `write` operations. Additionally, they don't use hard copy, and the result is reflected on the original `Matrix`.
- Reference: *The C++ Programming Language* by Bjarne Stroustrup.

Subscripting

Through `Matrix_slice` and `Matrix_ref`., the user is allowed to access elements through both *C-style* and *Fortran-style* subscripting.

```
template <typename... Dims>
size_t operator()(Dims... dims) const
{
    static_assert(sizeof...(Dims) == N, "Matrix_slice: unmatched subscripting
dimension.");

    size_t args[N]{size_t(dims)...}; // copy arguments into an array

    return start + std::inner_product(args, args + N, strides.begin(),
size_t(0));
}
```

We must do template specialization when `N=1` or `N=0`, to provide accessors that makes sense (instead of `mat[1][2][3]()` or any other explicit conversion).

```
// General Accessor
Matrix_ref<T, N - 1> row(size_t n)
{
    assert(n < (Matrix_base<T, N>::rows()));
    Matrix_slice<N - 1> row;
    Matrix_impl::slice_dim<0>(n, Matrix_base<T, N>::desc, row);
    // std::cerr << "(Matrix src row: " << Matrix_base<T, N>::desc.start << ")";
    return Matrix_ref<T, N - 1>(row, data());
}

// Specialization
T &row(size_t n)
{
    assert(n < (Matrix_base<T, 1>::rows()));
    // todo: desc access
    // std::cerr << "(Matrix ref row: " << Matrix_base<T, N>::desc.start << ")";
    return *(data() + Matrix_base<T, 1>::desc(n));
}

// Usage
cout << "The (1, 2, 1) element of mat2 with dimension 3 is " <<
mat2.row(1).row(2).row(1) << endl;
```

Now we give the implementation of the subscripting access. The *C-style* subscripting support is provided through the function `Matrix_ref<T, N - 1> row(size_t n)`. It is rather easy to see right now.

```
Matrix_ref<T, N - 1> operator[](size_t n) { return row(n); }
```

It is also specialized for `N=1` and `N=0`, and we shall not repeat them here. Now we have the *C-style* subscripting:

```
cout << mat2[1][2][1] << endl;
```

Now we turn to *Fortran-style* subscripting. Since we have `operator()` defined in `Matrix_slice`, we directly use that indexing.

```
template <typename... Dims>
Enable_if<Matrix_impl::Requesting_element<Dims...>(), T &> operator()(Dims...
dims)
{
    assert(Matrix_impl::check_bounds(Matrix_base<T, N>::desc, dims...));
    return *(data() + Matrix_base<T, N>::desc(dims...));
}
```

where `Enable_if` is a template alias for `std::enable_if`, and `check_bounds` is a template function that checks whether each of the dimension requested is out of bound:

```
// From namespace Matrix_impl
template <size_t N, typename... Dims>
bool check_bounds(const Matrix_slice<N> &slice, Dims... dims)
{
    size_t indexes[N]{size_t(dims)...};
    return std::equal(indexes, indexes + N, slice.extents.begin(),
std::less<size_t>());
}
```

Then we can directly access elements by:

```
    cout << mat2(1, 2, 1) << endl;
```

It is also possible to access like this:

```
    cout << mat2[1](2, 1) << endl;
    mat2[1][2][1] = 2333; // And do some modification!
```

because `mat[2]` retrieves the reference to the submatrix.

If the index is out of bound, we get an assertion failure.

```
T& utils::Matrix_ref<T, N>::operator()(Dims ...) [with Dims = {int, int}; T =
double; long unsigned int N = 2]: Assertion
`Matrix_impl::check_bounds(Matrix_base<T, N>::desc, dims...)' failed.
```

Region of Interest

We implement the **Region of Interest** function through `Matrix_slice` and `Matrix_ref`.

To implement ROI, we must first define an ***iterator*** (which is very important!), so that iterating through the contained elements with enhanced for is possible. It is also very convenient to use such iterator to construct a new `Matrix` based on an existing `Mat` object, regardless of whether it owns its elements (so that it has the right to destruct them and rearrange them).

Once a `Mat` object is created, the user has no way to redefine its dimension, nor to change its `extents`. In this way, modifying a `Matrix` object would not cause the defect of `Matrix_ref`s which attach to it.

We first give the implementation of the iterator of a `Mat` object, based on `Matrix_slice`.

```
/**
 * @brief Matrix_ref_iterator that provide access to member elements that can be
read/written.
 * @note The end of this iterator is determined by its degree, with the cursor
at the end having its maximum degree exceeded one more.
 *
 */
class Matrix_ref_iterator
{
    public:
    using iterator_category = std::forward_iterator_tag; // Tag can impact the
performance when used with STL algorithms
    using difference_type = std::ptrdiff_t;
    using value_type = T;
    using pointer = T *;
    using reference = T &;

    Matrix_ref_iterator() = default;
    Matrix_ref_iterator(Matrix_ref_iterator &&) = default;
    Matrix_ref_iterator &operator=(Matrix_ref_iterator &&) = default;
    Matrix_ref_iterator(Matrix_ref_iterator const &) = default;
    Matrix_ref_iterator &operator=(Matrix_ref_iterator const &) = default;
    ~Matrix_ref_iterator() = default;

    // Initialize
    Matrix_ref_iterator(Matrix_slice<N> &slice, T *start, std::array<size_t, N>
cursor = {})
        : slice(slice), start(start), ptr(start), cursor(cursor) {} //
Initialize with steps

    virtual reference operator*() const { return *ptr; }
    virtual pointer operator->() { return ptr; }

    // Prefix increment
    virtual Matrix_ref_iterator &operator++()
    {
        // adjusting cursor
        size_t step = N - 1;
        bool overflow = false;
        while (step >= 0)
        {
            if (++cursor[step] >= slice.extents[step])
            {
                if (step == 0)
                {
                    overflow = true;
                    break;
                }
                else
```

```
                {
                    cursor[step] = 0;
                    --step;
                }
            }
            else
                break;
        }

        // calculate offset
        size_t offset = std::inner_product(slice.strides.begin(),
slice.strides.end(), cursor.begin(), 0);
        ptr = start + offset;
        return *this;
    }

    // Postfix increment
    virtual Matrix_ref_iterator operator++(int)
    {
        Matrix_ref_iterator tmp = *this;
        ++(*this);
        return tmp;
    }

    friend bool operator==(const Matrix_ref_iterator &a, const
Matrix_ref_iterator &b) { return a.cursor == b.cursor; }
    friend bool operator!=(const Matrix_ref_iterator &a, const
Matrix_ref_iterator &b) { return a.cursor != b.cursor; }

    private:
    Matrix_slice<N> &slice;
    std::array<size_t, N> cursor; // store the current position in this Matrix,
0 initialized.
    pointer start = nullptr;
    pointer ptr = nullptr;
};

virtual Matrix_ref_iterator begin() { return Matrix_ref_iterator(desc, data());
}
virtual Matrix_ref_iterator end()
{
    std::array<size_t, N> m_end{};
    m_end[0] = desc.extents[0];
    return Matrix_ref_iterator(desc, data(), std::move(m_end));
}
```

- Currently, it is implemented through incrementing inner `cursor` (corresponding to its `extents`, therefore performance issue may occur.
- There should be other faster implementations, but using `std::inner_product` is very straight forward and easy to read.
- For plain `Matrix` object, usage of `std::vector<T>::iterator` is applied.

From this implementation, the following access method can be allowed:

```
// iterator test
cout << "Test normal iterator\n";
for (auto &i : mat2)
    cout << "Iterator test: " << i << "\n"; // Access the entire Matrix
cout << "========>OK.\n";

cout << "Test reference iterator\n";
for (auto &i : mat2[2])
    cout << "Iterator test: " << i << "\n"; // Access the second row
cout << "========>OK.\n";
```

`const_iterator`s are also implemented.

Through the usage of iterators, we can initialize `Matrix` and `Matrix_ref` from each other.

Element-by-Element Operation through `apply (F f)`

After having implemented the `iterator`, we can now finally turn to ELE operations.

```
template <typename F>
Matrix<T, N> &apply(F f)
{
    for (auto &x : (*this))
        f(x);
    return *this; // Enables chaining
}
```

This interface allows you to do something like

```
mat9.apply([&](int &a){ a += 10; });
```

to each element while also enabling other predefined per-element operations.

Other operations

All other operations (including per element arithmetic, multiplication,) can be implemented in the same way as in `Project 3`. Matrix multiplication is only defined for 2-D `Mat`s. However through subscripting and the usage of iterators, the user can quickly implement ***tensor*** multiplication that is suitable for N-dimensional operations. For details, please see source code.

Part 3 - Tests

Tests are constructed to mainly test whether the program will compile with different template parameters. Since we have only limited time to design this project, we only test the basic functions.

The test functions are built into the source file `test.cpp`, and since the list is too long, most of them is omitted.

```
// ...
```

```
Test reference iterator
Iterator test: 19
Iterator test: 20
Iterator test: 21
Iterator test: 22
Iterator test: 23
Iterator test: 24
Iterator test: 25
Iterator test: 26
Iterator test: 27
========>OK.
Test construction from Matrix_ref
Iterator test: 10
Iterator test: 11
Iterator test: 12
Iterator test: 13
Iterator test: 14
Iterator test: 17
Iterator test: 16
Iterator test: 2333
Iterator test: 18
3, 3
========>OK.
Test iterator change from Matrix
Iterator test: 11
Iterator test: 12
Iterator test: 13
Iterator test: 14
Iterator test: 15
Iterator test: 18
Iterator test: 17
Iterator test: 2334
Iterator test: 19
3, 3
========>OK.
Test per element apply from Matrix
Iterator test: 21
Iterator test: 22
Iterator test: 23
Iterator test: 24
Iterator test: 25
Iterator test: 28
Iterator test: 27
Iterator test: 2344
Iterator test: 29
3, 3
========>OK.
Test 2 per element apply from Matrix
Iterator test: 22
Iterator test: 23
Iterator test: 24
Iterator test: 25
Iterator test: 26
Iterator test: 29
Iterator test: 28
```

```
Iterator test: 2345
Iterator test: 30
3, 3
========>OK.

// ...
```

Part 4 - Difficulties

Writing a template class is extremely difficult for a beginner in C++. However, there are some solutions to certain problems that can be found by reading *The C++ Programming Language* by Bjarne Stroustrup or by endlessly searching about the issue on the Internet (which often end up in incorrect code, or code that are not elegant enough, as if it is a mistake to design the program in such way).

From my experience, it is way better to directly read code written by some famous programmers or organizations, but understanding the code requires much effort, as there is often no one providing guidance.

1. How to implement N-dimensional matrix?

   Solution. Using a template with argument `size_t N` to provide support.

2. How to deal with N-dimensional subscripting, recursive subscripting, etc.?

   Solution. We can implement subscripting through recursively fetch a row or column of the current `Mat` object (that is, either `Matrix_ref` or `Matrix` itself. In the following context, we shall call them interchangeably.)

   To implement recursive subscripting, we must define recursive template functions that correctly returns the `Mat` object with current `dimension`s, element access offset, column access offset, etc.

   We have to create a helper class called `Matrix_slice` to support accessing the `Mat` object with correct subscripting on its pointer (or `vector`), and transforming such an object requires some messy math formula. To provide access to element in the `Mat`, we also need to specialize the templates of `Mat` objects for `dimension=1` and `dimension=0`.

3. How to do partial specialization on functions of a template class provided that we must consider the performance issue when subscripting `Mat`s with lower order, and we must consider the case when arithmetic operations cannot be applied?

   Solution. `std::enable_if`, and `auto operator!=(const T&a, const T&b) -> decltype(!a==b);`.

   The latter is used to detect whether `T` supports comparing equality.

   - I googled `partial specialization` with different keywords, only to find that most users are saying that "you are not able to do partial specialization unless you partially specialize the whole template class".

4. How to efficiently apply element-by-element arithmetic operations to `Mat`s?

   Solution. By defining a template function called `apply(F f)` that accepts a function. In this way, it is easy to manage any operations that independently applies to each element of a `Mat` object. These functions must have **no side-effect**, though (in other words they must can be turned into lambda expressions).

5. Other metaprogramming problems and const qualifier problems (really, really disturbing, but also critical for developing correct codes. I have no guarantee that my code is completely correct, and I can only test limited samples.).