# Project 4: A Class to Describe a Matrix

**Sreyny THA-12113053**

## Part I: Analysis

**1. Support different data types:** Using Templates in C++ can support different types without explicitly specifying the type. In the case of the Matrix class, it is defined as a template class using the following syntax:

```
template <typename T>
class Matrix {
private:
    size_t rows;
    size_t cols;
    T* data;
    bool is_submatrix;
    size_t row_offset;
    size_t col_offset;
    Matrix* parent_matrix;
    }
}
```

The typename T in the angle brackets (< >) is a placeholder for the actual type that will be provided when creating an instance of the Matrix class. This means that T can be any type, such as int, float, double, or even a user-defined type. When we create an instance of the Matrix class, we specify the type by providing the desired type in the angle brackets. For example:

```
Matrix<int> mat1(3, 3);       // Matrix of integers
Matrix<float> mat2(3, 3);     // Matrix of floats
Matrix<double> mat3(3, 3);    // Matrix of doubles
Matrix<char> mat4(3, 3);    // Matrix of char
```

**2. Memory Management:** The Matrix class properly allocates and deallocates memory for the data array using new[] and delete[] in the constructors and destructor, respectively. The move constructor and move assignment operator also handle the memory correctly by setting the data pointer of the source object to nullptr to prevent double deletion. In addition, the assignment operator (operator=) handles the case where the source and destination matrices are the same object by checking if (this == &other) before performing any operations.

```
Matrix::~Matrix() {
    delete[] data;
}
```

```
    Matrix& operator=(Matrix&& other) noexcept {
        if (this == &other) return *this;
        if (!is_submatrix) {
            delete[] data;
        }
}
```

**3. Operation Overloading:** The Matrix class overloads operators such as assignment (=), equality (==), addition (+), subtraction (-), and multiplication ().

```
    Matrix& operator=(Matrix&& other) noexcept {
        if (this == &other) return *this;
        if (!is_submatrix) {
            delete[] data;
        }

        rows = other.rows;
        cols = other.cols;
        data = other.data;
        is_submatrix = other.is_submatrix;
        row_offset = other.row_offset;
        col_offset = other.col_offset;
        parent_matrix = other.parent_matrix;

        other.data = nullptr;
        return *this;
    }
}
```

```
    Matrix operator*(const Matrix& other) const {
        if (cols != other.rows) {
            throw std::invalid_argument("Number of columns in the first matrix
must match the number of rows in the second matrix for multiplication");
        }
        Matrix result(rows, other.cols);
        for (size_t i = 0; i < rows; ++i) {
            for (size_t j = 0; j < other.cols; ++j) {
                result(i, j) = 0;
                for (size_t k = 0; k < cols; ++k) {
                    result(i, j) += (*this)(i, k) * other(k, j);
                }
            }
        }
        return result;
    }
```

```
    bool operator==(const Matrix& other) const {
        if (rows != other.rows || cols != other.cols) {
            return false;
        }

        for (size_t i = 0; i < rows; ++i) {
            for (size_t j = 0; j < cols; ++j) {
                if ((*this)(i, j) != other(i, j)) {
                    return false;
                }
            }
        }
        return true;
    }
```

**4. Region of Interest (ROI):** The ROI allows sharing the same memory between two Matrix objects, where one object represents the entire matrix, and the other object represents a subregion of the matrix. The ROI is implemented using the following data members in the Matrix class: isROI, rowOffset, colOffset, roiRows, and roiCols. The setROI() function sets the ROI parameters.

To avoid memory hard copy, the Matrix class maintain a single data array for all instances, regardless of whether they represent the entire matrix or a subregion. The element access operators should take into account the ROI parameters to access the correct elements.

```
void Matrix::setROI(size_t rowOffset, size_t colOffset, size_t roiRows, size_t
roiCols) {
    if (rowOffset + roiRows > rows || colOffset + roiCols > cols) {
        throw std::out_of_range("ROI exceeds matrix dimensions.");
    }
    this->rowOffset = rowOffset;
    this->colOffset = colOffset;
    this->roiRows = roiRows;
    this->roiCols = roiCols;
    isROI = true;
}
```

```
int& Matrix::operator()(size_t row, size_t col) {
    return data[(row + rowOffset) * cols + (col + colOffset)];
}

const int& Matrix::operator()(size_t row, size_t col) const {
    return data[(row + rowOffset) * cols + (col + colOffset)];
}
```

## Part II: Result

```
int main() {

    Matrix<int> mat1(3, 3);
    for (size_t i = 0; i < 3; ++i) {
        for (size_t j = 0; j < 3; ++j) {
            mat1(i, j) = i * 3.0 + j;
        }
    }
    mat1.print();
    Matrix<float> mat2(3, 3);
    for (size_t i = 0; i < 3; ++i) {
        for (size_t j = 0; j < 3; ++j) {
            mat2(i, j) = 3.5f;
        }
    }
    mat2.print();
    Matrix<char> mat3(3, 3);
    for (size_t i = 0; i < 3; ++i) {
        for (size_t j = 0; j < 3; ++j) {
            mat3(i, j) = 'A';
        }
    }
    mat3.print();
    std::cout<<"SubMat: "<<std::endl;
    Matrix<int> submat = mat1.getSubmatrix(1, 1, 2, 2);
    submat.print();
    Matrix<int> result1 = mat1 + mat1;
    std::cout<<"Result1: "<<std::endl;
    result1.print();
    Matrix<float> result2 = mat2 * mat2;
    std::cout<<"Result2: "<<std::endl;
    result2.print();
    Matrix<char> result3 = mat3 + mat3;
    std::cout<<"Result3: "<<std::endl;
    result3.print();
    bool isEqual = (mat1 == result1);
    std::cout<<"isEqual: "<<isEqual<<std::endl;

    return 0;
}
```

**Output:**

```
0 1 2
3 4 5
6 7 8
3.5 3.5 3.5
3.5 3.5 3.5
3.5 3.5 3.5
A A A
A A A
```

```
A A A
SubMat:
4 5
7 8
Result1:
0 2 4
6 8 10
12 14 16
Result2:
36.75 36.75 36.75
36.75 36.75 36.75
36.75 36.75 36.75
Result3:
é é é
é é é
é é é
isEqual: 0
```