Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем

Расчетно-графическая работа
по дисциплине «Современные технологии программирования»
на тему «Реализация шаблонного типа данных Matrix»

Выполнила:
ст. гр. ИС-941
Степанищева А.В.

Проверил:
ст. пр. Пименов Е. С.

Новосибирск 2022

Содержание

1. Постановка задачи

Спроектировать шаблонный тип данных Matrix. Использовать стандарт языка C++17. Реализовать итераторы, совместимые с алгоритмами стандартной библиотеки. Покрыть модульными тестами. При реализации не пользоваться контейнерами стандартной библиотеки, реализовать управление ресурсами в идиоме RAII.

В качестве системы сборки использовать CMake. Структурировать проект в соответствии с соглашениями Canonical Project Structure [1]. Всю разработку вести в системе контроля версий git. Настроить автоматическое форматирование средствами clang-format.

Проверить код анализаторами Valgrind Memcheck, undefined sanitizer, address sanitizer, clang-tidy.

2. Мотивация

Матрица является одним из часто используемых математических объектов при решении различных практических задач во многих областях науки. К примеру, при решении задач линейной алгебры нередко сталкиваются с матричными вычислениями. В таких случаях использовать двумерный массив данных не всегда бывает удобным, так как приходится самостоятельно реализовывать большинство математических операций и, как следствие, возрастает вероятность допущения ошибок.

Структура данных Matrix – удобная замена двумерному массиву или вектору векторов. Матрица содержит в себе все плюсы такой структуры данных, как вектор и, кроме того, поддерживает операции умножения матриц, транспонирования и другие. Интуитивно очевидно, что наличие этих возможностей позволяет быстро и эффективно совершать различные вычисления над матрицами.

3. Реализация

3.1. Внутреннее устройство

Спроектированный шаблонный тип данных Matrix хранит элементы в одномерном массиве, что позволяет эффективнее обращаться к памяти за счет последовательного доступа к элементам.

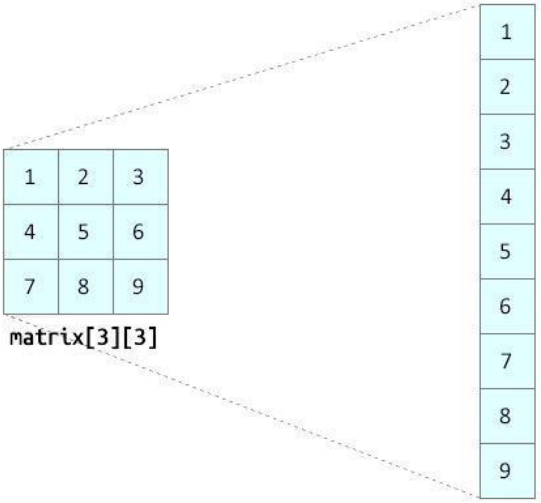

Рис. 1 Внутреннее устройство Matrix

3.2. Интерфейс

Rule of five

1. Matrix();
2. explicit Matrix(unsigned column, unsigned row);
3. explicit Matrix(unsigned column, unsigned row, std::vector<T> inputValues);
4. Matrix(const Matrix<T> &copyMatrix);
5. Matrix<T> &operator=(const Matrix<T> &copyMatrix);
6. Matrix(Matrix<T> &&moveMatrix) noexcept;
7. Matrix<T> &operator=(Matrix<T> &&moveMatrix) noexcept;
8. ~Matrix();

Строка 1. Конструктор по умолчанию. Инициализирует поле T *matrix_ как nullptr. Остальные приватные поля (column_, row_, matrixSize_ проинициализированы при объявлении). Сложность: константная.

Строка 2. Explicit конструктор, который принимает два параметра: количество строк и количество столбцов. Конструктор инициализирует поля переданными значениями, а матрица заполняется нулями. Сложность: линейная.

Строка 3. Explicit конструктор, который принимает три параметра: количество строк, количество столбцов и вектор значений, которыми должна быть заполнена матрица. Конструктор инициализирует поля переданными значениями, матрица заполняется значениями из вектора. Сложность: линейная.

Строка 4. Копирующий конструктор. Заполняет переданную матрицу значениями текущей. Сложность: линейная.

Строка 5. Copy assignment. Делает проверку на Self Assignment, после чего заполняет переданную матрицу значениями текущей, не выделяя дополнительной памяти. Сложность: линейная.

Строка 6. Перемещающий конструктор. Заполняет переданную матрицу значениями текущей, используя std::move. Сложность: линейная.

Строка 7. Move assignment. Заполняет переданную матрицу значениями текущей, не выделяя дополнительной памяти, используя std::move. Сложность: линейная.

Строка 8. Деструктор. Вызывает delete[] для матрицы. Сложность: линейная.

Member functions

1. unsigned getColumn() const;
2. unsigned getRow() const;
3. unsigned int getMatrixSize() const;
4. T *getMatrix() const;
5. const T operator()(unsigned rows, unsigned cols) const;
6. T &operator()(unsigned rows, unsigned cols);
7. Matrix<T> transpose();

Строки 1-4. Функции (геттеры), для получения таких значений как: количество столбцов и строк, размера матрицы и матрицы (указатель на матрицу). Сложность: константная.

Строки 5-6. Операторы, с помощью которых есть возможность получить доступ к элементу матрицы или присвоить значение элементу. Сложность: константная.

Строка 7. Метод, который возвращает транспонированную матрицу. Сложность: квадратичная.

Перегруженные операторы

Доступные через публичный интерфейс

1. Matrix<T> &operator+=(const Matrix<T> &bMatrix);
2. Matrix<T> &operator-=(const Matrix<T> &bMatrix);
3. Matrix<T> &operator*=(const T &scalar);

Строки 1-3. Выполняют операции: сложения и вычитания двух матриц, умножения матрицы на скаляр, и возвращает измененный объект. (Сложность: линейная.)

Другие операторы, которые поддерживает класс

1. std::ostream &operator<<(std::ostream &os, const Matrix<T> &outputMatrix)
2. bool operator==(const Matrix<T> &aMatrix, const Matrix<T> &bMatrix)
3. bool operator!=(const Matrix<T> &aMatrix, const Matrix<T> &bMatrix)
4. Matrix<T> operator+(const Matrix<T> &aMatrix, const Matrix<T> bMatrix)
5. Matrix<T> operator-(const Matrix<T> &aMatrix, const Matrix<T> bMatrix)
6. Matrix<T> operator*(const Matrix<T> &aMatrix, const Matrix<T> bMatrix)
7. Matrix<T> operator*(const Matrix<T> &aMatrix, T scalar)

Строка 1. Оператор вывода матрицы в консоль. (Сложность: квадратичная.)

Строка 2-3. Бинарные операторы, которые сравнивают две матрицы и возвращают true, если они равны (или неравны), а иначе возвращают false. (Сложность: линейная.)

Строка 4. Оператор сложения двух матриц, который возвращает результирующую матрицу сложения. (Сложность: линейная.)

Строка 5. Оператор вычитания двух матриц, который возвращает результирующую матрицу вычитания. (Сложность: линейная.)

Строка 6. Оператор умножения двух матриц, который возвращает результирующую матрицу умножения. (Сложность: кубическая.)

Строка 7. Оператор умножения матрицы на скаляр, который возвращает результирующую матрицу умножения. (Сложность: линейная.)

3.3. Итераторы

В рамках выбранной структуры данных был реализован random access iterator. Так как структура представлена как одномерный массив и, как следствие, имеет последовательное хранение данных в памяти, было принято решение реализовать двунаправленный итератор. Все операции совершаются за константное время.

1. reference operator*() const;
2. pointer operator->();
3. MatrixIterator &operator++();
4. MatrixIterator &operator—();
5. MatrixIterator operator++(int);
6. MatrixIterator operator—(int);
7. MatrixIterator operator+(size_t n) const;
8. MatrixIterator& operator+=(size_t n);
9. MatrixIterator operator-(size_t n) const;
10. MatrixIterator& operator-=(size_t n) const;
11. difference_type operator-(const MatrixIterator &other);
12. bool operator==(const MatrixIterator &other) const;
13. bool operator!=(const MatrixIterator &other) const;
14. bool operator>(const MatrixIterator &other) const;
15. bool operator<(const MatrixIterator &other) const;
16. bool operator>=(const MatrixIterator &other) const;
17. bool operator<=(const MatrixIterator &other) const;
18. reference operator[](size_t n) const;

difference_type = std::ptrdiff_t;

value_type = T;

pointer = value_type *;

reference = value_type &;

Строки 1-2. Операции для разыменования, чтобы получить значения, на которые указывает итератор.

Строки 3-6. Операции для инкрементирования и декрементирования, как в префиксной версии, так и в постфиксной, чтобы перемещать итератор на одну позицию вперед или назад.

Строки 7-10. Операции для смещения итератора на n позиций вперед или назад.

Строка 11. Операция возвращает расстояние между итераторами.

Строки 12-17. Операции сравнения с другим итератором.

4. Список источников

1. Kolpackov B. Canonical Project Structure [Электронный ресурс]. URL: https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1204r0.html (дата обращения: 07.05.2022).
2. Idioms for operator overloading [Электронный ресурс]. URL: https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading (дата обращения: 15.05.2022)
3. Matrix multiplication [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Matrix_multiplication (дата обращения: 15.05.2022)
4. Operator overloading [Электронный ресурс]. URL: https://isocpp.org/wiki/faq/operator-overloading#matrix-subscript-op (17.05.2022)
5. RAII [Электронный ресурс]. URL: https://csc-cpp.readthedocs.io/ru/2021/_static/lectures/03-raii.pdf (18.05.2022)
6. STL find end [Электронный ресурс]. URL: https://en.cppreference.com/w/cpp/algorithm/find_end (20.05.2022)
7. STL find [Электронный ресурс]. URL: https://en.cppreference.com/w/cpp/algorithm/find (20.05.2022)

5. Приложение

```
// file rgr/rgr/Matrix.hpp
#pragma once

#include <iostream>
#include <vector>

template <class T> class Matrix {

private:
  unsigned column_{0};
  unsigned row_{0};
  unsigned matrixSize_{0};
  T *matrix_;

  struct MatrixIterator {
    using iterator_category = std::random_access_iterator_tag;
    using difference_type = std::ptrdiff_t;
    using value_type = T;
    using pointer = value_type *;
    using reference = value_type &;

    explicit MatrixIterator(pointer ptr = nullptr) : ptr_(ptr) {}

    reference operator*() const { return *ptr_; }

    pointer operator->() { return ptr_; }

    MatrixIterator &operator++() {
      ptr_++;
      return *this;
    }

    MatrixIterator &operator--() {
      ptr_--;
```

```
  return *this;
}

MatrixIterator operator++(int) { // NOLINT
  MatrixIterator tmp = *this;
  ++(*this);
  return tmp;
}

MatrixIterator operator--(int) { // NOLINT
  MatrixIterator tmp = *this;
  --(*this);
  return tmp;
}

MatrixIterator operator+(size_t n) const {
  return MatrixIterator(ptr_ + n);
}

MatrixIterator& operator+=(size_t n) {
  ptr_ += n;
  return *this;
}

MatrixIterator operator-(size_t n) const {
  return MatrixIterator(ptr_ - n);
}

MatrixIterator& operator-=(size_t n) const {
  ptr_ -= n;
  return *this;
}

difference_type operator-(const MatrixIterator& iterator) {
  return ptr_ - iterator.ptr_;
```

```
    }

    bool operator==(const MatrixIterator &other) const {
      return ptr_ == other.ptr_;
    }

    bool operator!=(const MatrixIterator &other) const {
      return ptr_ != other.ptr_;
    }

    bool operator>(const MatrixIterator &other) const {
      return ptr_ > other.ptr_;
    }

    bool operator<(const MatrixIterator &other) const {
      return ptr_ < other.ptr_;
    }

    bool operator>=(const MatrixIterator &other) const {
      return ptr_ >= other.ptr_;
    }

    bool operator<=(const MatrixIterator &other) const {
      return ptr_ <= other.ptr_;
    }

    reference operator[](size_t n) const {
      return *(ptr_ + n);
    }

  private:
    pointer ptr_;
  };

public:
```

```
MatrixIterator begin() const { return MatrixIterator(matrix_); }

MatrixIterator end() const { return MatrixIterator(matrix_ + matrixSize_); }

unsigned getColumn() const { return column_; }

unsigned getRow() const { return row_; }

unsigned int getMatrixSize() const { return matrixSize_; }

T *getMatrix() const { return matrix_; }

Matrix();

Matrix(unsigned column, unsigned row);

Matrix(unsigned column, unsigned row, const std::vector<T> &inputValues);

Matrix(const Matrix<T> &copyMatrix);

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

Matrix(Matrix<T> &&moveMatrix) noexcept;

Matrix<T> &operator=(Matrix<T> &&moveMatrix) noexcept;

~Matrix() { delete[] matrix_; }

const T &operator()(unsigned rows, unsigned cols) const {
  if (rows >= row_ || cols >= column_) {
    throw std::out_of_range("&operator() : Index out of bounds");
  }
  return matrix_[column_ * rows + cols];
}
```

```
  T &operator()(unsigned rows, unsigned cols) {
    if (rows >= row_ || cols >= column_) {
      throw std::out_of_range("&operator() : Index out of bounds");
    }
    return matrix_[column_ * rows + cols];
  }

  Matrix<T> &operator+=(const Matrix<T> &bMatrix);

  Matrix<T> &operator-=(const Matrix<T> &bMatrix);

  Matrix<T> &operator*=(const T &scalar);

  Matrix<T> transpose();
};

template <class T> Matrix<T>::Matrix() : matrix_(nullptr) {}

template <class T>
Matrix<T>::Matrix(unsigned column, unsigned row) : column_(column), row_(row) {
  if (row_ == 0 || column_ == 0) {
    throw std::invalid_argument("Matrix constructor has 0 size");
  }
  matrixSize_ = column_ * row;
  matrix_ = new T[matrixSize_]();
}

template <class T>
Matrix<T>::Matrix(unsigned column, unsigned row,
                  const std::vector<T> &inputValues)
    : column_(column), row_(row) {
  if (row_ == 0 || column_ == 0) {
    throw std::invalid_argument("Matrix constructor has 0 size");
  }
  matrixSize_ = column_ * row_;
```

```
  matrix_ = new T[matrixSize_];

  std::copy(inputValues.begin(), inputValues.end(), matrix_);
}

template <class T>
Matrix<T>::Matrix(const Matrix<T> &copyMatrix)
    : column_(copyMatrix.column_), row_(copyMatrix.row_),
      matrixSize_(copyMatrix.matrixSize_), matrix_(new T[matrixSize_]) {

  std::copy(copyMatrix.begin(), copyMatrix.end(), matrix_);
}

template <class T>
Matrix<T> &Matrix<T>::operator=(const Matrix<T> &copyMatrix) {
  if (&copyMatrix == *this) {
    return *this;
  }
  Matrix tmp(copyMatrix);
  std::swap(tmp);
  return *this;
}

template <class T>
Matrix<T>::Matrix(Matrix<T> &&moveMatrix) noexcept
    : column_(std::move(moveMatrix.column_)), row_(std::move(moveMatrix.row_)),
      matrixSize_(std::move(moveMatrix.matrixSize_)),
      matrix_(std::move(moveMatrix.matrix_)) {
  moveMatrix.column_ = 0;
  moveMatrix.row_ = 0;
  moveMatrix.matrix_ = nullptr;
}

template <class T>
Matrix<T> &Matrix<T>::operator=(Matrix<T> &&moveMatrix) noexcept {
```

```
  Matrix tmp(std::move(moveMatrix));
  std::swap(tmp);
  return *this;
}

template <class T>
std::ostream &operator<<(std::ostream &os, const Matrix<T> &outputMatrix) {
  for (size_t i = 0; i < outputMatrix.getRow(); ++i) {
    for (size_t j = 0; j < outputMatrix.getColumn(); ++j) {
      os << outputMatrix.getMatrix()[outputMatrix.getColumn() * i + j] << " ";
    }
    os << "\n";
  }
  return os;
}

template <class T>
bool operator==(const Matrix<T> &aMatrix, const Matrix<T> &bMatrix) {
  if (aMatrix.getRow() != bMatrix.getRow() &&
      aMatrix.getColumn() != bMatrix.getColumn()) {
    throw std::invalid_argument("operator== : Different dimensions");
  }
  for (size_t i = 0; i < aMatrix.getMatrixSize(); ++i) {
    if (aMatrix.getMatrix()[i] != bMatrix.getMatrix()[i]) {
      return false;
    }
  }
  return true;
}

template <class T>
bool operator!=(const Matrix<T> &aMatrix, const Matrix<T> &bMatrix) {
  if (aMatrix.getRow() != bMatrix.getRow() &&
      aMatrix.getColumn() != bMatrix.getColumn()) {
    throw std::invalid_argument("bool operator!= : Different dimensions");
```

```
  }
  for (size_t i = 0; i < aMatrix.getMatrixSize(); ++i) {
    if (aMatrix.getMatrix()[i] != bMatrix.getMatrix()[i]) {
      return true;
    }
  }
  return false;
}

template <class T>
Matrix<T> operator+(const Matrix<T> &aMatrix, const Matrix<T> bMatrix) {
  if (aMatrix.getRow() != bMatrix.getRow() &&
      aMatrix.getColumn() != bMatrix.getColumn()) {
    throw std::invalid_argument("operator+ : Different dimensions");
  }
  Matrix<T> matrixProduct(aMatrix.getRow(), aMatrix.getColumn());
  for (size_t i = 0; i < aMatrix.getMatrixSize(); ++i) {
    matrixProduct.getMatrix()[i] =
        aMatrix.getMatrix()[i] + bMatrix.getMatrix()[i];
  }
  return matrixProduct;
}

template <class T> Matrix<T> &Matrix<T>::operator+=(const Matrix<T> &bMatrix) {
  if (row_ != bMatrix.row_ && column_ != bMatrix.column_) {
    throw std::invalid_argument("operator+= : Different dimensions");
  }
  for (size_t i = 0; i < matrixSize_; ++i) {
    matrix_[i] += bMatrix.matrix_[i];
  }
  return *this;
}

template <class T>
Matrix<T> operator-(const Matrix<T> &aMatrix, const Matrix<T> bMatrix) {
```

```
  if (aMatrix.getRow() != bMatrix.getRow() &&
      aMatrix.getColumn() != bMatrix.getColumn()) {
    throw std::invalid_argument("operator- : Different dimensions");
  }
  Matrix<T> matrixProduct(aMatrix.getRow(), aMatrix.getColumn());
  for (size_t i = 0; i < aMatrix.getMatrixSize(); ++i) {
    matrixProduct.getMatrix()[i] =
        aMatrix.getMatrix()[i] - bMatrix.getMatrix()[i];
  }
  return matrixProduct;
}

template <class T> Matrix<T> &Matrix<T>::operator-=(const Matrix<T> &bMatrix) {
  if (row_ != bMatrix.row_ && column_ != bMatrix.column_) {
    throw std::invalid_argument("operator+= : Different dimensions");
  }
  for (size_t i = 0; i < matrixSize_; ++i) {
    matrix_[i] -= bMatrix.matrix_[i];
  }
  return *this;
}

template <class T>
Matrix<T> operator*(const Matrix<T> &aMatrix, const Matrix<T> bMatrix) {
  if (aMatrix.getColumn() != bMatrix.getRow()) {
    throw std::invalid_argument("operator* : The number of A-matrix column has "
                                "to be equal to the number of"
                                "B-matrix row");
  }
  Matrix<T> matrixProduct(bMatrix.getColumn(), aMatrix.getRow());

  for (size_t i = 0; i < aMatrix.getRow(); ++i) {
    for (size_t j = 0; j < bMatrix.getColumn(); ++j) {
      for (size_t k = 0; k < aMatrix.getRow(); ++k) {
        matrixProduct(i, j) += aMatrix(i, k) * bMatrix(k, j);
```

```
      }
    }
  }
  return matrixProduct;
}

template <class T> Matrix<T> operator*(const Matrix<T> &aMatrix, T scalar) {
  unsigned size = aMatrix.getRow() * aMatrix.getColumn();
  Matrix<T> matrixProduct(aMatrix.getColumn(), aMatrix.getRow());
  for (size_t i = 0; i < size; ++i) {
    matrixProduct.getMatrix()[i] = aMatrix.getMatrix()[i] * scalar;
  }
  return matrixProduct;
}

template <class T> Matrix<T> &Matrix<T>::operator*=(const T &scalar) {
  unsigned size = getRow() * getColumn();
  for (size_t i = 0; i < size; ++i) {
    getMatrix()[i] *= scalar;
  }
  return *this;
}

template <class T> Matrix<T> Matrix<T>::transpose() {
  Matrix<T> matrixProduct(column_, row_);
  for (size_t i = 0; i < row_; ++i) {
    for (size_t j = 0; j < column_; ++j) {
      matrixProduct(i, j) = matrix_[column_ * j + i];
    }
  }
  return matrixProduct;
}
```