CS205 C/ C++ Program Design - Assignment 4

Please declare a class `Matrix` in file matrix.hpp and implement its member functions and friend functions (if any) in file matrix.cpp. How the matrix works is demonstrated in main.cpp with a main() function inside.

Requirements:

1. (20 points) The class `Matrix` should be designed for a matrix with `float` elements. It should contain some members to describe its number of rows and columns, data and others. Please use `private` keyword to protect the data inside.
2. (35 points) Please design some constructors, destructor, operator =, operator << and some others as what ***Lecture Notes in Week 11*** describe. Since the data of a matrix is normally large, please do not use data copy in constructors and operator =. You can follow the design of `cv::Mat` in OpenCV.
3. (25 points) Please implement operator * overloading such that the class can support: C = A * B, C = A * b, and C = a * B (Capital letters are for matrices. Small letters are for scalars).
4. (10 points) Compile and run your program on an ARM development board.
5. (5 points) Please host your source code at GitHub.com,
6. (5 points) Please use cmake to manage your source code.

Rules:

1. Please submit your assignment report before its deadline. After the deadline (even 1 second), 0 score!
2. Please pay more attention to your code style. After all this is not ACM-ICPC contest. You have enough time to write code with both correct result and good code style. You will get deduction if your code style is terrible. You can read Google C++ Style Guide (http://google.github.io/styleguide/cppguide.html) or some other guide for code style.

CS205 C/ C++ Program Design

Assignment 4

Name: 王奕童(YeeTone Wang)

SID: 11910104

Instructor: 于仕琪(Shiqi Yu)

Part 1. Description

本次Assignment4一共上交5个文件：

1. main.cpp -->测试主程序
2. Data.h -->封装数据的类的头文件
3. Data.cpp -->封装数据的类的实现文件
4. CMatrix.h -->基于C++语言的矩阵的头文件
5. CMatrix.cpp -->基于C++语言的矩阵的实现文件

注1：本次上交的文件的矩阵乘法的实现中，为了精简只保留了最基本的实现方法，去除了大部分多余冗杂的优化策略。
注2：本次上交的文件不依赖于特定的指令集，因此具有较好的跨平台运行的优势。

【Requirement0】
用于数据封装的Data类的基本结构：

```
9   class Data {
10  private:
11      float* values= nullptr;
12      int referenceTimes=0;
13  public:
14      explicit Data(long size);
15      ~Data();
16      [[nodiscard]] int getReference()const;
17      float* get_values_ptr();
18      void addReference();
19      void removeReference();
20      void setDefaultRef();
21  };
```

【Requirement1】

CMatrix类内部封装关于数据的3个成员变量均用private修饰符以控制访问权限。
3个成员变量类型：

```
private:
    //Requirement1:
    int row=0;//the Row number
    int column=0;//the Column number
    Data* dataptr= nullptr;//the float elements pointer
```

【Requirement2】
实现了构造函数(constructor)，析构函数(destructor)，以及Lecture11 Notes中要求的<<和=运算符。其中第三种构造函数CMatrix(const CMatrix& cm)，以及=运算符的重载的实现借鉴了OpenCV的实现，将两个对象的内容通过增加数据引用次数的方法指向同一片内存区域，以避免空间浪费。

1. 构造函数

```
    //Requirement2: constructor
    explicit CMatrix(int r=0,int c=0);
    explicit CMatrix(int r,int c,const float* outData);
    CMatrix(const CMatrix& cm);
```

2. 析构函数

```
    //Requirement2: destructor
    ~CMatrix();
```

3. 运算符<<和=

```
    //Requirement2: operator << and =
    friend ostream& operator<<(ostream& os,const CMatrix& cm);
    CMatrix& operator=(const CMatrix& cm);
```

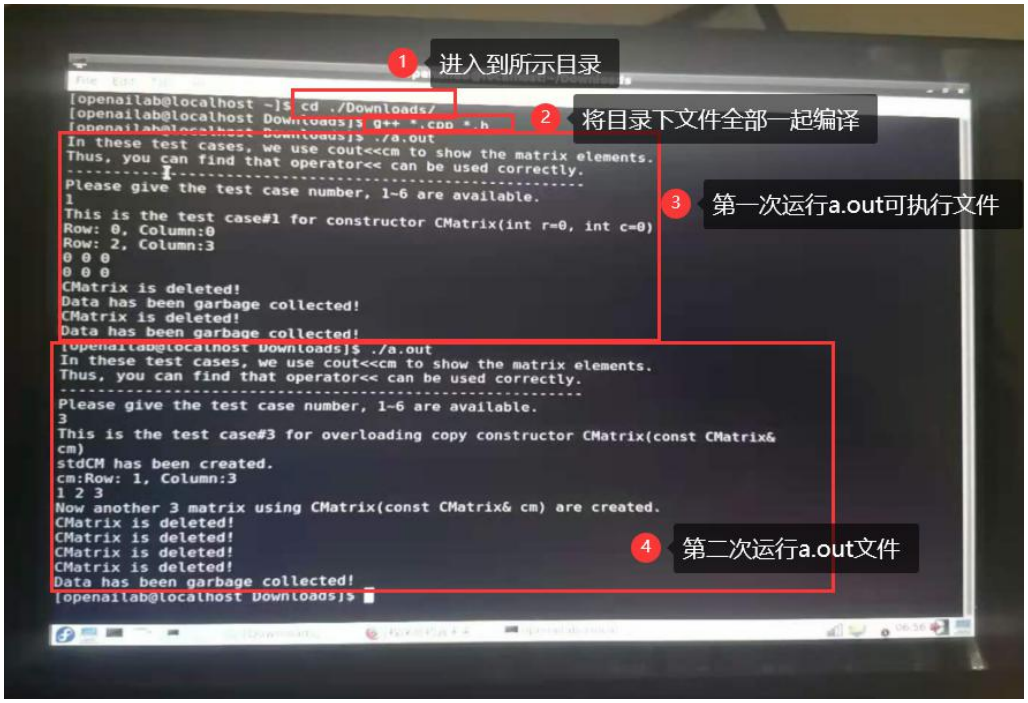
4. 内存管理

```
    void data_gc();
```

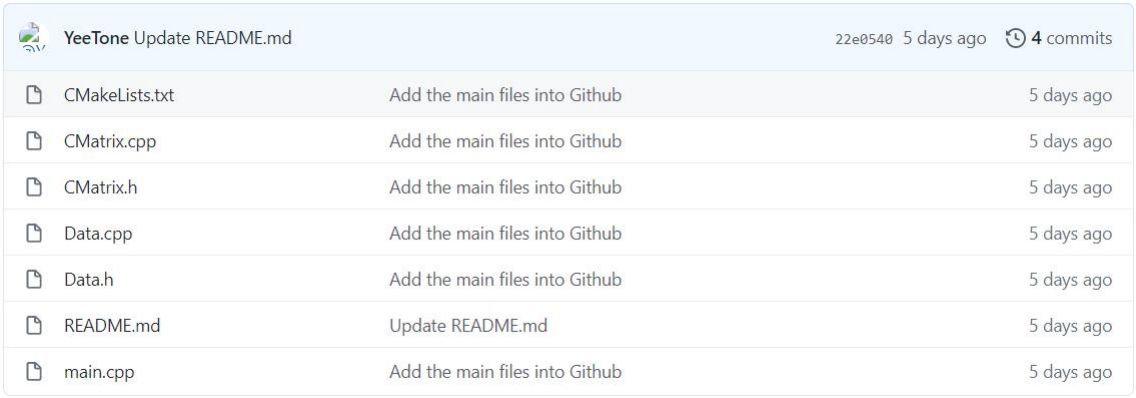
【Requirement3】
关于矩阵乘法，实现了5种功能的运算符：
C=A*B，C=A*b，C=a*B，A*=B和A*=b
(其中大写字母A, B代表矩阵，而小写字母a, b代表标量(数字))

```
    CMatrix operator*(const CMatrix& cm) const;
    friend CMatrix operator*(const CMatrix& cm,int x);
    friend CMatrix operator*(int x,const CMatrix& cm);
    void operator*=(const CMatrix& cm);
    void operator*=(int x);
```

【Requirement4】
在ARM开发板上实现编译和运行



【Requirement5】
上传源码至Github
Github仓库链接：https://github.com/YeeTone/CS205Assignment4



【Requirement6】
实现使用cmake来管理源码，该功能通过Clion实现。
请见与Github仓库里的CMakeList.txt
【Additional Implementation】
额外运算符实现：
矩阵加法：

```
27  CMatrix operator+(const CMatrix& cm) const;
37  void operator+=(const CMatrix& cm);
```

矩阵减法：

```
28  CMatrix operator-(const CMatrix& cm) const;
38  void operator-=(const CMatrix& cm);
```

矩阵内容是否相等：

```
35  friend bool operator==(const CMatrix& cm1,const CMatrix& cm2);
36  friend bool operator!=(const CMatrix& cm1,const CMatrix& cm2);
```

通过下标获取矩阵元素：

```
41      const float* operator[](int i);
```

矩阵快速幂:

```
33      CMatrix operator^(int x) const;
```

额外功能实现:
通过下标获取矩阵元素:

```
43      [[nodiscard]] float get(int r,int c) const;
```

获取矩阵的行和列:

```
45      [[nodiscard]] int getRow() const;
46      [[nodiscard]] int getColumn() const;
```

获取矩阵的数据内容的引用次数:

```
47      [[nodiscard]] int getRef()const;
```

允许进行矩阵的克隆和复制:

```
48      [[nodiscard]] CMatrix clone()const;
49      void copyOf(const CMatrix& cm);
```

Part 2. Result & Verification

There are **6** test cases in total to test how the main function of matrix works, and these 6 cases are available in main.cpp.

【Case1】test for constructor `CMatrix(int r, int c);`

```
Please give the test case number, 1~6 are available.
1
This is the test case#1 for constructor CMatrix(int r=0, int c=0)
Row: 0, Column:0
Row: 2, Column:3
0 0 0
0 0 0
CMatrix is deleted!
Data has been garbage collected!
CMatrix is deleted!
Data has been garbage collected!
```

1 调用了无参构造方法，因此矩阵内没有元素

2 调用了两个参数的构造方法，并令r=2，c=3，因此矩阵为两行三列，内部元素默认置零

3 子函数运行完毕，自动回收临时变量的空间，因此调用了矩阵的析构函数和数据的垃圾回收

【Case2】test for constructor `CMatrix(int r, int c, const float *outData);`

```
Please give the test case number, 1~6 are available.
2
This is the test case#2 for constructor CMatrix(int r,int c,const float* outData)
Row: 2, Column:3
1 2 3
4 5 6
CMatrix is deleted!
Data has been garbage collected!
```

1 先设置一个默认float指针，指向数组[1,2,3,4,5,6]

2 调用该3个参数的构造方法，为了数据的保护性，该构造方法采取的是内存copy的策略。

3 函数运行完毕，临时变量销毁，调用析构函数与数据回收

【Case3】test for overloading copy constructor

CMatrix(const CMatrix &cm);

```
Please give the test case number, 1~6 are available.
3
This is the test case#3 for overloading copy constructor CMatrix(const CMatrix& cm)
stdCM has been created.
cm:Row: 1, Column:3
1 2 3
Now another 3 matrix using CMatrix(const CMatrix&
CMatrix is deleted!
CMatrix is deleted!
CMatrix is deleted!
CMatrix is deleted!
Data has been garbage collected!
```

1 创建一个数据为[1,2,3]的矩阵

2 调用3次copy constructor来建立3个相同的矩阵对象

3 函数运行结束，标准矩阵和3个临时变量都被回收，因此有4行矩阵的删除信息；但由于指向的是同一片内存区域，数据的回收只被调用了1次，故只有1行数据的回收信息，也说明数据能够被正常地回收处理掉。

【Case4】test for destructor ~CMatrix(); and operator=

```
Please give the test case number, 1~6 are available.
4
This is the test case#4 for destructor ~CMatrix() and operator=
Now stdMatrix 3x2 has been created!
Row: 2, Column:3
1 2 3
4 5 6
Data has been garbage collected!
Data has been garbage collected!
Data has been garbage collected!
3 Matrixes using = has been created as well!
CMatrix is deleted!
CMatrix is deleted!
CMatrix is deleted!
CMatrix is deleted!
Data has been garbage collected!
```

1 构建一个两行三列的矩阵，内部元素为1.0f~6.0f

2 先构建3个默认参数矩阵，内部没有元素；然后使用连等号赋值cm1=cm2=cm3=stdCM; 因此cm1,cm2,cm3的数据空间都被回收，打印3行数据回收信息

3 函数运行结束，4个CMatrix的临时变量都被回收，因此打印4行矩阵删除信息

4 但由于4个CMatrix对象都指向同一片数据内存区域，因此回收的时候只回收了1次，打印了1次回收信息

【Case5】 test for operator* for A*B

```
Please give the test case number, 1~6 are available.
5
This is the test case#5 for operator* overloading: A*B
CMatrix1:Row: 2, Column:3
1 2 3
4 5 6
CMatrix2:Row: 3, Column:2
1 2
3 4
5 6
CMatrix is deleted!
Row: 2, Column:2
22 28
49 64

CMatrix is deleted!
Data has been garbage collected!
CMatrix is deleted!
Data has been garbage collected!
CMatrix is deleted!
Data has been garbage collected!
```

1 构建1个2x3的矩阵，内部元素是1.0f~6.0f

2 构建1个3x2的矩阵，内部元素同为1.0f~6.0f

3 cout<<cm1*cm2<<endl;
此处在运算过程中产生了一个临时变量[redacted]，生命周期仅限于乘法函数的内部，因此会优先被销毁，因此先显示了删除信息，后显示运算结果。

4 函数体运行结束，矩阵乘法的结果以及两个原矩阵的内存和数据全被回收处理，程序正常退出

【Case6】 test for operator* for A*b and a*B

```
Please give the test case number, 1~6 are available.
6
This is the test case#6 for operator* overloading: A*b and a*B
cmatrix:Row: 2, Column:3
1 2 3
4 5 6
------------
2*cmatrix:Row: 2, Column:3
2 4 6
8 10 12
CMatrix is deleted!
Data has been garbage collected!
------------
cmatrix*2:Row: 2, Column:3
2 4 6
8 10 12
CMatrix is deleted!
Data has been garbage collected!
CMatrix is deleted!
Data has been garbage collected!
```

1 构建一个2x3的矩阵

2 测试2*cmatrix

3 创建的表示2*cmatrix的匿名变量很快就被销毁

4 测试cmatrix*2

5 创建的表示cmatrix*2的匿名变量很快就被销毁

6 函数体结束，一开始构建的cmatrix变量被销毁

Part 3. Difficulties & Solutions, or others

【Description】copy constructor和operator=的实现中，参考了C++11中新添加的智能指针的实现策略，建立了Data类用于封装数据类型为float指针从而避免大型数据拷贝和错误的内存回收。

【Background】C++11中添加的智能指针：

头文件要求：`#include <memory>`

可使用的指针类：

`std::shared_ptr`：多个指针指向相同的对象，使用引用计数策略

`std::unique_ptr`：仅限1个unique_ptr指向对象，离开作用域时自动执行delete

`std::weak_ptr`：仅能进行观测操作，不具有普通指针的行为

【Solution】参考OpenCV，python以及智能指针的垃圾回收策略，建立一个class用于封装float指针和引用次数，如下所示：

```
class Data {
private:
    float* values= nullptr;
    int referenceTimes=0;
public:
    explicit Data(long size);
    ~Data();
    [[nodiscard]] int getReference()const;
    float* get_values_ptr();
    void addReference();
    void removeReference();
    void setDefaultRef();
};
```

然后在CMatrix类里面定义一个Data类型的指针指向该Data对象即可：、

```
class CMatrix {
private:
    //Requirement1:
    int row=0;//the Row number
    int column=0;//the Column number
    Data* dataptr= nullptr;//the float elements pointer
```

主要需要考虑以下情形：

1. 开新矩阵的时候数据的建立与存储
2. 外部传入float数据指针用于初始化时的内存拷贝
3. copy constructor和operator=的实现中，释放原先指向的内存区域并指向传入的内存区域的管理
4. 矩阵销毁的时候的数据回收

1. 开新矩阵的时候数据的建立与存储

开新矩阵的时候，先尝试进行数据的回收（一般回收不到，因为对于空指针回收不到内存，此处是为了安全起见）将矩阵所需要的Data类的数据空间通过new关键字申请即可。

```
    data_gc();
    this->dataptr=new Data( size: r*c);
```

那么data_gc()方法的实现的源码如下：

```
286 void CMatrix::data_gc(){
287     if(this->dataptr== nullptr){
288         return;
289     }
290     if(this->dataptr->getReference()<=1){
291         cout<<"Data has been garbage collected!"<<endl;
292         delete this->dataptr;
293     }else{
294         this->dataptr->removeReference();
295     }
296 }
```

①如果是空指针的话，就不进行回收，避免报错;
②如果Data指针指向的对象的引用次数小于等于1（这个对象是最后一个使用该片数据的对象），则启动Data指针的回收处理，通过最后的析构函数回收掉所有内部封装的float指针指向的内存;

```
12 Data::~Data() {
13     delete values;
14 }
```

③其他情况，则是Data指针指向的对象的引用次数不少于2次的情形（还有其他矩阵对象在使用该片数据），则移除一次引用次数，使得引用次数减少1。

```
28 void Data::removeReference() {
29     this->referenceTimes-=1;
30 }
```

2. 外部传入float数据指针用于初始化时的内存拷贝
同上，先执行data_gc()的内存回收，然后再将Data指针开出新的内存空间，最后调用memcpy函数进行快速的内存值复制即可。

```
21     data_gc();
22     this->dataptr=new Data( size: r*c);
23     memcpy( _Dst: this->dataptr->get_values_ptr(),outData, _Size: ((long)sizeof(float))*r*c);
```

3. copy constructor和operator=的实现中，释放原先指向的内存区域并指向传入的内存区域的管理
先判断即将被处理的矩阵是否和传入的矩阵是同一个对象，如果是同一个对象的话，就不需要再额外考虑复制处理;

```
26     if(this==(&cm)){
27         return;
28     }
```

否则就考虑先复制一些int变量(row, column)回收内存，然后指向同一片内存区域，最后增加指向区域的引用计数即可。

```
29      this->row=cm.row;
30      this->column=cm.column;
31      data_gc();
32      this->dataptr=cm.dataptr;
33      this->dataptr->addReference();
```

4. 矩阵销毁的时候的数据回收

在析构函数时直接调用data_gc()的内部方法即可，根据指向区域的引用次数来决定是直接释放内存还是减少引用次数。

```
35  CMatrix::~CMatrix() {
36      cout << "CMatrix is deleted!" << endl;
37      data_gc();
38  }
```

【Description】后续尝试增加对于矩阵个数的描述变量：

```
14  private:
15      static int count;
16      int matrix_id;
```

并修改相关的构造方法和析构方法，增加矩阵id的监测输出：

3个构造方法最后都添加了以下3条语句：

```
18      count++;
19      matrix_id=count;
20      cout<<"CMatrix"<<count<<" is created!"<<endl;
```

析构方法添加以下语句：

```
46      cout << "CMatrix"<<matrix_id<<" is deleted!" << endl;
47      count--;
```

然后执行测试用例4，输出结果如下：

```
This is the test case#4 for destructor ~CMatrix() and operator=
CMatrix1 is created!
Now stdMatrix 3x2 has been created!
Row: 2, Column:3
1 2 3
4 5 6
CMatrix2 is created!
CMatrix3 is created!
CMatrix4 is created!
Data has been garbage collected!
Data has been garbage collected!
Data has been garbage collected!
3 Matrixes using = has been created as well!
CMatrix4 is deleted!
CMatrix3 is deleted!
CMatrix2 is deleted!
CMatrix1 is deleted!
Data has been garbage collected!
```

我们可以看到创建的次序是1234，然后销毁的次序是4321。

【Reason Analysis】

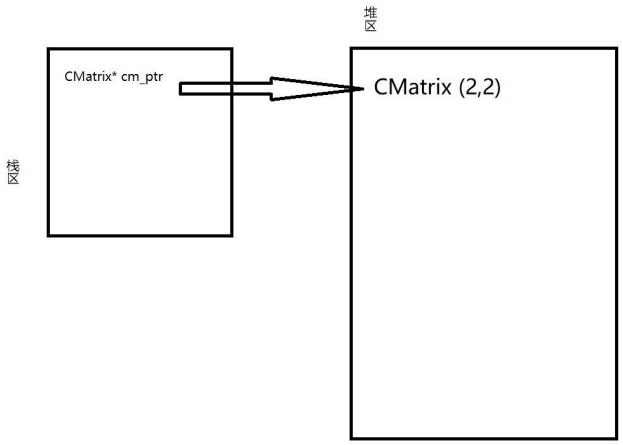
C/C++语言编译的程序一般有以下3个部分：

1. 栈区：用于存放函数的参数值，局部变量的值，由编译器自动分配释放，自动实现内存管理与回收；
2. 堆区：由运行程序实现分配释放，需要手动管理内存；程序运行结束时被操作系统所回收；
3. 全局区：用于存储静态变量和全局变量，程序结束后被操作系统所回收；
4. 常量区：用于存储常值变量，如字符串等。该区域数据一旦初始化就不能被修改，运行结束后被操作系统所回收

在函数体(test_case4)的内部，都采用的是调用构造方法来实例化对象，这样生成的对象都存储在栈区里面。而栈的特性是先入后出，后入先出，因此就会显示出来创建次序为1234，而销毁次序为4321的结果。

另外，在一句使用new语句申请内存的时候，cm_ptr是存储在栈里面的，而new申请出来的内存是在堆里面的，并且不一定是一段连续的内存。cm_ptr是一个在栈区，但是指向堆内存的指针，如下所示：

```
CMatrix* cm_ptr=new CMatrix( r: 2, c: 2);
```



Part4.Summary

本次Assignment4基于CS205课程的期中矩阵乘法project，主要目的是融会贯通C/C++的运算符重载的特性，并加深对于C/C++语言对于内存的管理的理解。本次作业要求明确，在做作业的过程中学习了很多的知识点：

1. 类的数据封装特性；
2. 对于C/C++语言中指针特性的理解；
3. C/C++的类的copy constructor和destructor的理解；
4. C++语言中类运算符重载的实现；
5. C++语言中友元运算符的重载的实现；
6. C/C++语言中指针指向内存数据区域的管理；
7. C++11中添加的智能指针的理解与使用；
8. C/C++在不同平台上的运行效果的差异；
9. 对于Github代码共享平台的使用；
10. 对于在ARM开发板上编译运行C/C++语言的功能的使用；
11. 对于计算机操作系统的结构的理解与认识；