CS205 C/C++ Programming - Project 3

Name: 钟元吉(Zhong Yuanji)

SID: 12012613

Part 1 - Analysis

> This project is to implement only `float` elements matrix structure and some relavant functions only by using `C` language.

本次项目在要求只使用`C`语言，建立矩阵结构体及实现构造、复制、删除、矩阵间及矩阵与数的加法、减法、乘法、最大最小值、矩阵行列式、求逆等数学计算函数，其中需要考虑的点较多。在设计这样的矩阵操作库时，我们需要考虑以下问题：

1. 检查函数中传入的参数是否合理
2. 无法使用C++中的`#program once`，如何防止头文件重复包含导致的重定义
3. 如何记录返回结果的同时返回错误信息
4. 如何快速地进行矩阵求行列式与矩阵求逆
5. 如何从字符串中读取矩阵

以及在不同系统下运行产生的问题(见Part 4 问题1)。

1. 检查函数中传入的参数是否合理

考虑到使用者在使用函数是可能会将空指针作为参数传入函数，或者在取子矩阵和取余子式矩阵时输入了越界的行数或列数，也有可能在没有判断行列式不为0的时候将一个不可逆的矩阵求逆，因此，我们的程序需要对这些错误进行一一检验。

对输入的代码进行以下类型的错误检验：

错误类型	对应输出整数
运算中输入的第一个矩阵或其数据是空指针	-1
运算中输入的第二个矩阵或其数据是空指针	-2
运算中输出矩阵是空指针	-3
运算中输入的两个矩阵行数或列数错误，不满足运算要求	-4
取子矩阵或余子式时输入的行数或列数越界或有误	-5
求逆矩阵时，矩阵不可逆	-6

注意：在判断出现错误后，函数将不会对输出矩阵`ret`进行赋值。

前三种错误类型可以归纳为对于传入参数指针非空的要求，其中，传入未初始化的指针也是不允许的，例如，下面的做法是禁止的：

```
Matrix *mat; // 禁止传入未初始化的指针
float num = det(mat);
```

这里，我们可以通过4种方式初始化：

```
Matrix *mat1 = NULLMatrix;
Matrix *mat2 = createMatrix(1, 2, (__f *)malloc(2 * __SIZEF));
Matrix *mat3 = createMatrixFromStr("[1,2;3 4]");
Matrix *mat4 = (Matrix *)malloc(__SIZEM);
mat4->row = 1;
mat4->col = 2;
mat4->data = (__f *)malloc(2 * __SIZEF));
```

2. 如何防止头文件重复包含导致的重定义

在C++中，我们一般会在头文件开始时使用`#program once`来防止头文件重复包含。在C中，由于`#program once`是C++在C的基础上增加的内容，我们无法使用。因此我们在`.h`的头文件开始时，用将点变为下划线的文件名做为变量名，在**未定义过的条件下进行宏定义**，并在该条件下定义其他内容。如果之前已经导入该文件，或者关联到已经导入该头文件的`.o`文件时，其中一个将不再起作用。

进一步我们会在宏名称的前后加上双下划线，以防止出现重复。

3. 如何记录返回结果的同时返回错误信息

我们知道，在C和C++中，函数无法像在`matlab`中一样，同时返回多个值。因此我们可以考虑改变传入指针或引用对应的数据，以达到返回结果的目的。引用也是C++在C的基础上增加的内容，我们无法在C中使用，因此我们将函数**传入参数的最后一个作为返回结果的指针**。

但是，这也意味着存在一个问题，函数无法对这个矩阵结构体指针重新分配内存，也就是无法使这个指针指向新的矩阵结构体，于是，我们**要求传入的输出矩阵结构体指针不能是空指针**，要求输出矩阵结构体的浮点数数组指针不能指向非零的无效位置（因为非零时无法判断是否有效）。

4. 如何快速地进行矩阵求行列式与矩阵求逆

矩阵求行列式的方法有许多，但是每种方法有不同的时间复杂度。例如，我们可以用逆序数的方法，计算出$n!$项n个数相乘的展开，但是这样的的时间复杂度为$O(nn!)$。我们也可以使用某一行的余子式展开成n个`n-1`阶行列式来进行计算，这样计算的复杂度为$O(n!)$。而如果我们使用**高斯消元法**，将矩阵化简为上三角矩阵，然后取对角线相乘得到行列式，这样计算的时间复杂度为$O(n^3)$，这远远比前面的两种方法耗时短。我们只需在遇到$M_{i,i}=0$时寻找$M_{j,i}\neq 0, j>i$然后交换i,j两行，若$M_{j,i}=0, \forall j>i$，则这时行列式为0。

5. 如何从字符串中读取矩阵

为了更方便地构造矩阵，我们提供了从字符串到矩阵的构造方法，我们规定输入矩阵需要按照`matlab`中矩阵的输入格式，使用`[]`将矩阵包括，用英文逗号或前面无逗号、分号的空格作为列分割符，（前面有逗号、分号的空格将被忽略），用英文分号作为行分隔符（矩阵末尾`]`前，多余的逗号、分号、空格将被忽略）。对于不满足格式的字符串输入将会返回空矩阵`NULLMatrix`，例如：

```
Matrix *mat5 = createMatrixFromStr("[1, 2;3 4; ]");
printf(to_string(mat5));
```

则会输出：

```
Matrix 2x2:
[
   1.000000e+00   2.000000e+00
   3.000000e+00   4.000000e+00
]
```

实现过程：

1. 复制一份字符串。
2. 去除多余的空格，如果空格前为空格、逗号、分号将被去除，否则换为逗号。
3. 去除字符串末尾的右括号、逗号和分号。
4. 从第二个字符开始，记录逗号和分号个数，对应计算出行数和列数。
5. 初始化矩阵结构体，并逐行读取写入，并释放复制的字符串内存。
6. 如果出现错误，释放内存，返回空矩阵。

Part 2 - Code

由于代码较长，这里仅放置**矩阵求行列式**及**矩阵求逆**的部分代码，其他函数简介请参考源文件`MatrixFunc.c`或函数头文件`Matrix.h`，源文件中的注释比较详细。

注：将`float`类型定义为`__f`，如果需要使用`double`等其他类型时，便于转换。

```
// Compute the determinant of the matrix
__f det(const Matrix *mat)
{
    // Check if exist and rows equals columns
    if (!mat || mat->row == 0 || mat->col != mat->row)
        return NULLF;
    // Copy the data
```

```
    int row = mat->row, col = mat->col;
    __f *data1 = mat->data, ret = 1;
    __f *data = (__f *)malloc(row * col * __SIZEF);
    for (int i = 0; i < row * col; ++i)
        data[i] = data1[i];
    // Eliminate for each row using Gauss Method
    for (int i = 0; i < row; ++i)
    {
        // Check M_{i,i} != 0, if not change the rows
        for (int j = i; j < row; ++j)
            if (data[j * col + i] != 0)
            {
                if (i != j)
                { // Exchange two rows, determinant
                    for (int k = i; k < col; ++k)
                    {
                        __f tmpF = data[i * col + k];
                        data[i * col + k] = data[j * col + k];
                        data[j * col + k] = tmpF;
                    }
                    if ((j - i) % 2)
                        ret *= -1;
                }
                break;
            }
        // If no rows can make M_{i,i} != 0, the determinant is 0
        if (data[i * col + i] == 0)
            return 0;
        // Eliminate by using Gauss Method
        ret *= data[i * col + i];
        for (int j = i + 1; j < row; ++j)
        {
            __f num = data[j * col + i] / data[i * col + i];
            for (int k = i; k < col; ++k)
                data[j * col + k] -= num * data[i * col + k];
        }
    }
    // Delete the copy and return
    free(data);
    return ret;
}

// Compute the inverse of the matrix
int inv(const Matrix *mat, Matrix *ret)
{
    // Check if exist and rows equals columns
    __CheckMatRet;
    if (mat->row == 0 || mat->row != mat->col)
        return -4;
    __f matDet = det(mat);
    if (matDet == 0)
        return -6;
    // If the size of matrix is 1x1
    if (mat->row == 1)
    {
```

```
        if (!ret->data || ret->row != 1 || ret->col != 1)
        {
            if (ret->data)
                free(ret->data);
            ret->row = 1, ret->col = 1;
            ret->data = (__f *)malloc(__SIZEF);
        }
        ret->data[0] = 1 / mat->data[0];
        return 1;
    }
    // Else compute by using algebraic complement
    __RetMat(row, col,
             Matrix *tmp = NULLMatrix;
             cofactorMatrix(mat, j, i, tmp);
             data_[i * col + j] = ((i + j) % 2 ? -det(tmp) : det(tmp)) / matDet;
             deleteMatrix(tmp);
             ;);
    return 1;
}
```

Part 3 - Result & Verification

Test case #1: 基本要求的实现

注：程序中统一采用科学计数法进行输出矩阵对应的字符串

文件Demo1.c：

```
#include <stdio.h>
#include <stdlib.h>
#include "inc/Matrix.h"
// #include "src/MatrixFunc.c"
// For output
#define __show(n, mat)             \
    temp = to_string(mat);         \
    printf("mat%d = %s", n, temp); \
    if (mat->row && mat->col)      \
        free(temp);
// Main method
int main()
{
    char *temp; // For freeing
    Matrix *mat1 = NULLMatrix;
    __show(1, mat1);
    Matrix *mat2 = createMatrix(1, 2, (__f *)malloc(2 * __SIZEF));
    __show(2, mat2);
    Matrix *mat3 = createMatrixFromStr("[1, 2;3 4; ]");
    __show(3, mat3);
    printf("=======================================\n");
    copyMatrix(mat3, mat1);
    __show(1, mat1);
    deleteMatrix(mat3);
    mat3 = createMatrixFromStr("[1,2,3;4,5,6;7,8,9]");
```

```
    __show(3, mat3);
    subMatrix(mat3, 1, 3, 1, 3, mat2);
    __show(2, mat2);
    cofactorMatrix(mat3, 1, 1, mat1);
    __show(1, mat1);
    printf("=======================================\n");
    // mat1 = [1,3;7,9], mat2 = [5,6;8,9];
    addMatrix(mat1, mat2, mat3);
    __show(3, mat3);
    subtractMatrix(mat1, mat2, mat3);
    __show(3, mat3);
    addScalar(mat2, 0.1, mat3);
    __show(3, mat3);
    subtractScalar(mat2, 0.1, mat3);
    __show(3, mat3);
    multiplyMatrix(mat1, mat2, mat3);
    __show(3, mat3);
    multiplyScalar(mat2, 0.5, mat3);
    __show(3, mat3);
    printf("=======================================\n");
    // mat1 = [1,3;7,9], mat2 = [5,6;8,9];
    printf("determinant of mat1 = %f\n", det(mat1));
    printf("determinant of mat2 = %f\ninverse of mat1 = ", det(mat2));
    inv(mat1, mat3);
    __show(3, mat3);
    multiplyMatrix(mat1, mat3, mat2);
    __show(2, mat2);
    printf("=======================================\n");
    // mat1 = [1,3;7,9];
    transpose(mat1, mat2);
    __show(2, mat2);
    rotate90(mat1, mat3);
    __show(3, mat3);
    printf("minimal of mat3 = %f\n", minOfMatrix(mat3));
    printf("maximal of mat3 = %f\n", maxOfMatrix(mat3));
    return 0;
}
```

通过动态链接库编译运行：

```
gcc -shared -fPIC ./src/MatrixFunc.c -o libMatrix.so
gcc ./Demo1.c -o Demo1 -L. libMatrix.so
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH # Linux
./Demo1
```

Windows系统上：去掉第三行，第四行加上后缀`.exe`

输出：

```
mat1 = Matrix 0x0: []
mat2 = Matrix 1x2:
[
   0.000000e+00   0.000000e+00
]
mat3 = Matrix 2x2:
```

```
[
   1.000000e+00   2.000000e+00
   3.000000e+00   4.000000e+00
]
=======================================
mat1 = Matrix 2x2:
[
   1.000000e+00   2.000000e+00
   3.000000e+00   4.000000e+00
]
mat3 = Matrix 3x3:
[
   1.000000e+00   2.000000e+00   3.000000e+00
   4.000000e+00   5.000000e+00   6.000000e+00
   7.000000e+00   8.000000e+00   9.000000e+00
]
mat2 = Matrix 2x2:
[
   5.000000e+00   6.000000e+00
   8.000000e+00   9.000000e+00
]
mat1 = Matrix 2x2:
[
   1.000000e+00   3.000000e+00
   7.000000e+00   9.000000e+00
]
=======================================
mat3 = Matrix 2x2:
[
   6.000000e+00   9.000000e+00
   1.500000e+01   1.800000e+01
]
mat3 = Matrix 2x2:
[
  -4.000000e+00  -3.000000e+00
  -1.000000e+00   0.000000e+00
]
mat3 = Matrix 2x2:
[
   5.100000e+00   6.100000e+00
   8.100000e+00   9.100000e+00
]
mat3 = Matrix 2x2:
[
   4.900000e+00   5.900000e+00
   7.900000e+00   8.900000e+00
]
mat3 = Matrix 2x2:
[
   2.900000e+01   3.300000e+01
   1.070000e+02   1.230000e+02
]
mat3 = Matrix 2x2:
[
   2.500000e+00   3.000000e+00
```

```
   4.000000e+00   4.500000e+00
]
=======================================
determinant of mat1 = -12.000000
determinant of mat2 = -3.000002
inverse of mat1 = mat3 = Matrix 2x2:
[
  -7.500000e-01   2.500000e-01
   5.833333e-01  -8.333334e-02
]
mat2 = Matrix 2x2:
[
   1.000000e+00   0.000000e+00
   0.000000e+00   1.000000e+00
]
=======================================
mat2 = Matrix 2x2:
[
   1.000000e+00   7.000000e+00
   3.000000e+00   9.000000e+00
]
mat3 = Matrix 2x2:
[
   3.000000e+00   9.000000e+00
   1.000000e+00   7.000000e+00
]
minimal of mat3 = 1.000000
maximal of mat3 = 9.000000
```

图片较长，与上面的结果相同，不在此附图。

Test case #2: 对错误输入的判断

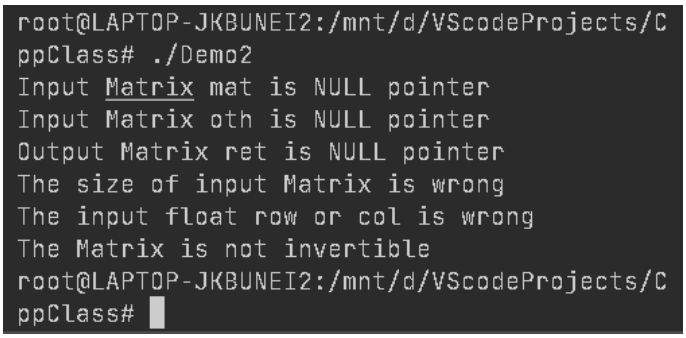
文件Demo2.c：

```c
#include <stdio.h>
#include <stdlib.h>
#include "inc/Matrix.h"
// #include "src/MatrixFunc.c"
// Error tips
const char *ERR_STR[] = {"",
     "Input Matrix mat is NULL pointer\n",
     "Input Matrix oth is NULL pointer\n",
     "Output Matrix ret is NULL pointer\n",
     "The size of input Matrix is wrong\n",
     "The input float row or col is wrong\n",
     "The Matrix is not invertible\n"};
// Main method
main()
{
    Matrix *mat1 = createMatrixFromStr("[0]");
    Matrix *mat2 = createMatrix(1, 2, (__f *)malloc(2 * __SIZEF));
    Matrix *mat3 = NULLMatrix;
```

```
    Matrix *mat4 = NULL;
    int code = inv(mat4, mat2);
    printf(ERR_STR[-code]);
    code = addMatrix(mat1, mat4, mat2);
    printf(ERR_STR[-code]);
    code = subtractScalar(mat1, 2, mat4);
    printf(ERR_STR[-code]);
    code = multiplyMatrix(mat2, mat1, mat3);
    printf(ERR_STR[-code]);
    code = subMatrix(mat1, -1, 1, 0, 1, mat3);
    printf(ERR_STR[-code]);
    code = inv(mat1, mat2);
    printf(ERR_STR[-code]);
    return 0;
}
```

输出：

```
Input Matrix mat is NULL pointer
Input Matrix oth is NULL pointer
Output Matrix ret is NULL pointer
The size of input Matrix is wrong
The input float row or col is wrong
The Matrix is not invertible
```



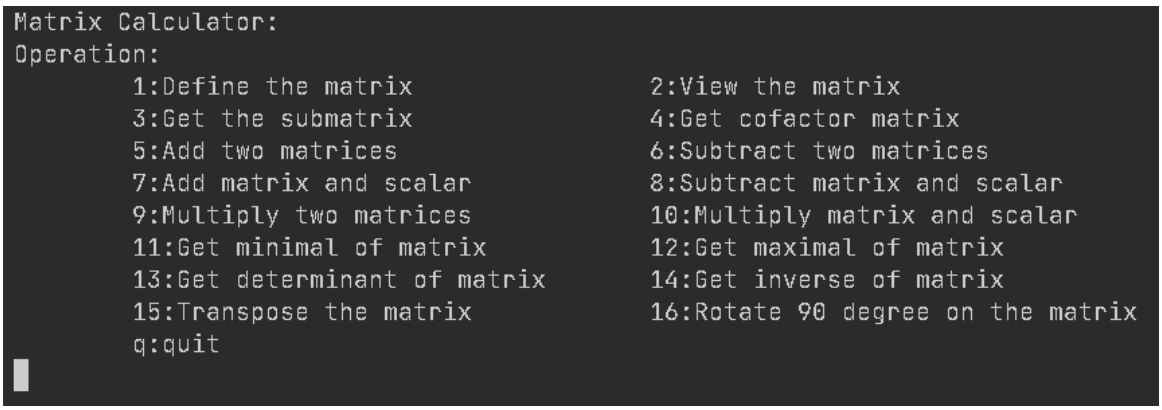
Test case #3: 程序特色功能：交互式的矩阵计算器

通过`makefile`编译运行：

```
make
```

输出提示：

```
Matrix Calculator:
Operation:
        1:Define the matrix                     2:View the matrix
        3:Get the submatrix                     4:Get cofactor matrix
        5:Add two matrices                      6:Subtract two matrices
        7:Add matrix and scalar                 8:Subtract matrix and scalar
        9:Multiply two matrices                 10:Multiply matrix and scalar
        11:Get minimal of matrix                12:Get maximal of matrix
        13:Get determinant of matrix            14:Get inverse of matrix
        15:Transpose the matrix                 16:Rotate 90 degree on the matrix
        q:quit
```



程序内可自由测试，在Part 4问题2中将举一个简单例子，其他不做赘述。

Part 4 - Difficulties & Solutions

问题1：在不同系统下运行产生的问题

由于使用者使用的计算机操作系统不尽相同，我们的代码需要兼容不同的操作系统。对于已知的Windows中MinGW32编译器，以及Linux中的Ubuntu GNU编译器，其中指针的大小不同。在MinGW使用的32位编译器中，指针的大小为4字节；在Linux的64位操作系统中，指针的大小为8字节。为了防止不同的操作系统上出现矩阵结构体的内存申请不足及过多的情况，这里使用宏定义来判断操作系统，在32位的Windows操作系统中已经将`__SIZEOF_POINTER__`定义为4了，因此我们可以根据`#if __SIZEOF_POINTER__ == 4`是否定义来判断是否是32位操作系统，进而定义正确的结构体内存大小。当然使用`sizeof(Matrix)`也能做到一样的效果，但是在每次构造新结构体的时候都会重新计算结构体大小，这样会增大计算量。前面这个例子可以通过判断指针的大小来定义结构体大小，而下面这个例子与操作系统关系更大。

在设计交互式矩阵计算器时，需要用到清屏的指令，这个指令在Windows操作系统中是`"cls"`，这个指令在Linux操作系统中却是`"clear"`，如果我们没有识别正确的操作系统，导致使用了错误的指令，进而会造成运行报错。因此，我们需要在宏定义中识别操作系统是否为Linux，根据`__linux__`是否已经定义，我们可以定义指令的字符串为`"clear"`或`cls`，进而实现不同操作系统上的清屏操作。

问题2：为了更方便地使用其中的大部分功能，如何设计交互式的矩阵计算器

考虑到通过代码使用矩阵结构体相关的函数需要手动开辟内存存储数据，再传入构造方法，使用函数时需要判断输出整数是否为1再进行输出，输出时需要使用`printf`以及手动释放字符串内存，较为麻烦。所以，提供一个可交互的矩阵计算器(或称矩阵操作系统)是很有必要的。

通过输入1~16来进行一次矩阵的赋值、查看或运算操作，输入0~9来选择自定义的矩阵序号，通过输入不带空格的字符串来构造矩阵（因为空格在`scanf`传入时会被认作分隔符），每步操作后均会判断是否有错误并立即打印对应的错误提示。

在矩阵计算器设计的过程中有许多重复的内容，程序中使用带双下划线前缀的宏定义函数进行统一化，减少代码长度，使代码更易读。

矩阵计算器的具体实现过程：

1. 清屏、输出提示
2. 分析输入的1~16对应的矩阵操作
3. 获取指定的输入矩阵编号
4. 进行矩阵操作并提示是否成功，若失败，提示错误信息
5. 若成功，显示结果或获取指定的输出矩阵编号进行保存

下面演示几个简单的例子，输入(将其中的空格改为换行，输出只放了结果部分)：

```
Input: 1 1 [1,2,3;1,4,5;1,6,9]
Output: Matrix 3x3:
        [
           1.000000e+00   2.000000e+00   3.000000e+00
           1.000000e+00   4.000000e+00   5.000000e+00
           1.000000e+00   6.000000e+00   9.000000e+00
        ]
Input: b 14 1 2
Output: The answer is: Matrix 3x3:
        [
           1.500000e+00  -0.000000e+00  -5.000000e-01
          -1.000000e+00   1.500000e+00  -5.000000e-01
           5.000000e-01  -1.000000e+00   5.000000e-01
        ]
Input: b 9 1 2
Output: The answer is: Matrix 3x3:
        [
           1.000000e+00   0.000000e+00   0.000000e+00
           0.000000e+00   1.000000e+00   0.000000e+00
           0.000000e+00   0.000000e+00   1.000000e+00
        ]
Input: b 5 0 1
Output: Operation failed!
        Error: Input Matrix mat is NULL pointer
```

问题3：如何使矩阵的乘法效率更高

在程序涉及的矩阵操作中，除了行列式和求逆，矩阵与矩阵的乘法是运算最大的操作，它的时间复杂度为$O(n^3)$。由于行数和列数的不确定性，我们很难在算法上进行复杂度的优化，但是我们可以改变以前的部分思路，做到一定程度的优化。

正常的矩阵乘法是使用$i-j-k$循环的，也就是说对于$C_{m\times r}=A_{m\times n}B_{n\times r}$中的每个元素满足：

$$C_{ij}=\sum_{k=1}^{n}A_{ik}B_{kj}$$

固定最外面两层的循环，即固定i, j，则C_{ij}被固定，如果有C++的引用，可以将C_{ij}的别名放在k所在循环的外面，循环内对这个别名进行加法赋值，这样可以减少寻找地址取数的次数；在C中只有指针，将一个指针代替C_{ij}仍然不可避免地需要寻址取数，效率提高不大。

而这里我们考虑**使用$i-k-j$的循环**，($j-k-i$与之类似)，则固定最外面两层的循环时，固定了i, k，其中不作为赋值左边的A_{ik}被固定，我们可以用一个局部变量将其存储，进而避免对A_{ik}的重复寻址取值，从而提升效率。

Part 5 - Summary

本次项目设计矩阵结构体库，从便于使用者调用的角度考虑，提供了从字符串构造矩阵的方法，同时提供了交互式矩阵计算器的一种实现；从运行环境不同的角度考虑，进行了对操作系统的宏定义判断；从数据类型统一的角度考虑，将数据类型`float`宏定义为`__f`，使后续更换为`double`或`long double`更方便。有些时候，程序不一定要使用很多较为复杂的算法，更应该便于使用、便于后续的修复。

以上是本次Report的所有内容，感谢您的阅读！