CS205 C/C++ Programming - Project3 Report

Name: 罗皓予（Haoyu Luo）

SID：12112517

Part 0 - Catalogue

3.3 释放矩阵

4.运算与极值

4.1 找矩阵中的最值

4.2 矩阵与标量运算

4.3 矩阵加减法

4.4 矩阵乘法

4.5 矩阵转置，单位矩阵

4.6 矩阵行列式，检查奇异矩阵

4.7 矩阵求逆

5.其他方法

1.随机生成矩阵数据

2.违法矩阵检查

6.程序的健壮性

1.文件读写的健壮性

2.初始化的健壮性

3.操作函数的健壮性

Part 3 - Result & Verification

1.三种方法生成浮点数矩阵

2.复制，释放和删除矩阵

3.矩阵求极值

4.矩阵标量计算

5.访存硬件级优化对矩阵乘法速度对比

6.矩阵求行列式

7.矩阵求逆

Part 4 - Difficulties&Solutions

1.函数调用复制结构体速度慢问题及解决

2.函数返回指向自动变量内存的指针问题及解决

3.无法在函数内删除矩阵指针问题及解决

4.无法编译内联函数的问题及解决

Part 5 Conclusion

Part 1 - Description

1. GitHub链接

本次项目源码将存于Github仓库。

(https://github.com/jimmylaw21/CS205-C-C-Program-Design/tree/main/project3)

2. 源码结构

本次作业共上交3个文件：

1.source1.c (main.c)

2.source2.c (mat)

3.header.h (mat.h)

3. 源码概述

3.1 关于main源码的解释说明

由于本次project的主要任务是设计一个易用的matrix结构体，因此本程序的main函数并未承载太多任务，仅保有测试功能，里面有七个测试程序，其运行结果已收录在报告的Part 3 - Result & Verification。

3.2 关于mat源码头文件的解释说明

0.学习链接：

Leo-Adventure/CS205-CPP-projects: The repository contains all five projects implemented in Sustech CS205 C/C++ class (github.com)

Amoiensis/Matrix_hub: A lib of Matrix operation for C language. (矩阵运算库--C语言) (github.com)

1.支持多种方式创建矩阵和填充矩阵

2.支持深拷贝，释放和删除矩阵

3.支持矩阵与标量加减乘除和求矩阵内元素最值

4.支持矩阵加减乘法

5.支持矩阵转置，计算行列式，求逆

Part 2 - Analysis

1. 矩阵基本实现

1.1 头文件分析

为了使得整体代码的风格简洁便于调试，此次project延续了上一个project的风格使用了头文件来进行函数原型的声明，以及矩阵结构的实现。头文件可以使本次project中各部分程序之间保持信息的一致性，同时将引入#ifndef宏来保护头文件，若整个代码当中如果多次引用了该头文件，则只有一次会生效，生效后会定义一个宏变量，并在之后不会再进入被保护的代码区域，这样就能防止头文件在编译程序时被重复引用。

1.2 结构声明

在本次project中，一个矩阵头结构体含有三个属性，分别是它的行数（row）以及列数（col），以及指向矩阵数据地址的float类型指针（data）。由于我们通常储存的矩阵巨大，因此该结构体不会将数组数据定义在内，同时由于是指针属性，矩阵数据大小可以动态分配，不必调整结构体内的矩阵数据大小，使用指针益处多多。声明代码如下：

```
typedef struct Matrix {
  int row;
  int col;
  float *data;
}Mat;
```

在该结构体的声明中，还使用关键字typedef可以为类型起一个新的别名，将struct Matrix精简为Mat，这种写法更加简练，意义也非常明确，不管是在标准头文件中还是以后的编程实践中，都会大量使用这种别名。

typedef 在表现上有时候类似于 #define，但它和宏替换之间存在一个关键性的区别。正确思考这个问题的方法就是把 typedef 看成一种彻底的“封装”类型，声明之后不能再往里面增加别的东西。typedef 和 #define有以下区别：1) 可以使用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这样做；2）在连续定义几个变量的时候，typedef 能够保证定义的所有变量均为同一类型，而 #define 则无法保证。

但是，方便的同时，我们也要小心使用 typedef 带来的陷阱，接下来看一个简单的 typedef 使用示例，如下面的代码所示：

```
typedef char* PCHAR;
int strcmp(const PCHAR,const PCHAR);
```

在上面的代码中，“const PCHAR” 是否相当于 “const char*” 呢?

答案是否定的，原因很简单，typedef 是用来定义一种类型的新别名的，它不同于宏，不是简单的字符串替换。因此，“const PCHAR”中的 const 给予了整个指针本身常量性，也就是形成了常量指针“char*const*（*一个指向char的常量指针*）”。*即它实际上相当于“char*const”，而不是“const char*（*指向常量 char 的指针*）”。当然，要想让 const PCHAR 相当于 const char* 也很容易，如下面的代码所示：

```
typedef const char* PCHAR;
int strcmp(PCHAR,  PCHAR);
```

1.3 内存分配

在C语言当中，使用malloc函数实现内存的动态分配。通过查看该函数的原型，得知函数的返回类型是void*指针类型，所以在实际返回的时候需要使用目标的指针类型对返回的指针类型进行强制转换。malloc函数返回的是一段以字节为单位分配的一段内存块，所以需要使用一个Mat类型的指针来追踪这一段分配的内存。分配方式如下：

```
Mat *mat1= (Mat *)malloc(sizeof(Mat));
```

由于矩阵内部有一段指向矩阵内元素的数组还没有分配内存，所以在调用初始化矩阵方法的时候也需要为该数组动态分配内存。

```
mat->nums= (float*)malloc(r*c*sizeof(float));
```

由于矩阵是float类型的矩阵，总共的元素个数为行数乘以列数，通过使用C语言的运算符sizeof(float)得到每一个float类型元素所需要分配的字节数，得出给一个行数为r，列数为c的矩阵分配内存总共需要分配的字节总数为r * c * sizeof(float)。

1.4 C/C++对比——struct

C语言中：Struct是用户自定义数据类型（UDT）。

C++语言中：Struct是抽象数据类型（ADT），支持成员函数的定义。

在C语言当中对于结构体无法在内部定义结构体初始化方法以及对本结构体进行操作的方法，而在C++当中可以这样实现。

C中的struct只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数。为低成本,低功耗,高性能服务。

在C++当中，结构体相比C语言更像是一个类，C++的struct可以当作class来用，里面可以实现声明内部数据以及对数据的操作。

在标准C++中，struct和class有两个区别：
第一：struct中的成员默认是public的，class中的默认是private的。
第二：在用模版的时候只能写template < class Type>或template < typename Type>不能写template < struct Type>。

在C语言如果要初始化一个结构体中的各个数据则需要重新编写方法对其初始化，对内部数据的操作方法的编写也限制较大，这也是两种语言的不同点之一。

1.5 C/C++对比——动态内存分配

在C++当中，实现动态分配内存使用new运算符后接需要分配内存的类型即可实现内存的分配，new 运算符会自动地分配指定类型的内存，也自动计算需要分配内存的字节数目。

同时，new申请的内存块不被释放，也会造成内存泄露，C++主要使用delete来释放new申请的内存。delete用于释放new申请的单个元素分配的内存，delete[]用于释放new []申请的多个元素分配的内存。要注意的是，使用delete释放的是内存块，而不是s这个指针变量，释放完后，s就变为随机值了，如果在接下来的程序中，没有给s赋新值就再调用s，由于s在内存中是随机值，就有可能指向重要地址，以致使系统崩溃。为了程序的健壮性，如果delete之后程序还没结束，就将s的值赋为NULL。

C语言使用的是malloc函数，返回的是已分配内存的首地址，并没有明确的类型，所以需要使用强制类型转换获得目标类型的指针，更为底层地，C语言在动态分配内存的时候需要输出分配的字节数，由于可能存在字节计算出错的问题，所以在动态分配内存方面，使用C++编写程序也更加不容易出错。同时在错误处理上，若申请失败，new会抛出bac_alloc异常，而malloc会返回NULL。

同时，C语言还使用了calloc函数和realloc函数。calloc也用于内存分配，它申请完内存块后,calloc将该内存块存储的值初始化为0，然后才返回指向该内存块起始位置的指针，另外两个函数请求内存大小的方式不同，详见函数原型：

```
void *calloc(size_t num_elements,size_t element_size);
int *p = (int *)calloc(5,sizeof(int));
```

realloc主要用于修改已经分配的内存的大小，函数原型为

```
void *realloc(void *ptr,size_t new_size);
```

ptr代表已经分配内存的首地址，new_size代表修改后的大小。若用于扩大一个内存块，该内存块原先的内容保留，新增加的内存添加到原先的内存块的后面，并且新增加的部分没有初始化；若用于缩小一个内存块，该内存块的尾部的部分内存被裁减掉，前面的部分内存的内容依然保留。

另外，若原先的内存块的大小无法改变，realloc将分配另一块指定大小的内存块，并将原先内存块的内容复制到新的内存块上。若发生这种情况，内存块的首地址就发生了改变，因此应使用新的指针接收realloc函数的返回值。

2.文件存取

2.1 C语言文件存取原理

程序与数据的交互是以流的形式进行的，进行C语言文件的存取时，都会先打开数据流，完成操作之后便需要关闭数据流。

C语言的文件处理功能依据系统是否设置“缓冲区”分为两种：一种是设置缓冲区，另一种是不设置缓冲区。由于不设置缓冲区的文件处理方式，必须使用较低级的I/O函数(包含在头文件io.h和fcntl.h中)来直接对磁盘存取，这种方式的存取速度慢，并且由于不是C的标准函数，跨平台操作时容易出问题。

当使用标准I/O函数（如本程序中使用的<stdio.h>头文件）进行文件读取时，系统会自动设置缓冲区，并通过数据流读写文件。即当进行文件读取时，不会直接对磁盘进行读取，而是先打开数据流，将磁盘上的信息拷贝到缓冲区内，程序再从缓冲区读取所需的数据。当写入文件时，并不会马上写入磁盘中，而是先写入缓冲区，只有在缓冲区已满或“关闭文件”时，才会将数据写入磁盘。

2.2 C语言数据流的打开与关闭

通过查看stdio.h头文件里面关于文件读取函数的内容，发现打开文件的函数原型为

```
/* Open a file and create a new stream for it.
This function is a possible cancellation point and therefore not
marked with __THROW. */
extern FILE *fopen (const char *__restrict __filename,const char *__restrict
__modes) __wur;
```

该函数具有两个形参，一个是文件名，一个是打开文件的模式，返回类型是一个FILE类型的指针。在本程序中涉及到的打开文件模式有"r" 和 "w"， "r"：只能从文件中读数据，该文件必须先存在，否则打开失败，"w"：只能向文件写数据，若指定的文件不存在则创建它，如果存在则先删除它再重建一个新文件。

在使用fopen函数之前先要声明了FILE指针指向打开的文件，如果文件打开失败则会返回空指针，所以可以根据返回的指针是否非空判定文件是否成功打开。

```
int fclose(FILE *stream);
```

在操作结束之后，需要及时关闭数据流，如果没有及时关闭数据流，尽管在程序结束之后会自动关闭所有数据流，但文件打开过多会导致系统运行缓慢，这时就要自行手动关闭不再使用的文件，来提高程序整体的执行效率。关闭数据流可以通过调用 fclose函数实现。如果成功关闭，则会返回非零值。

2.3 C语言文件读写

文件的类型分为文本文件和二进制文件。文本文件是以字符编码的方式进行保存的，二进制文件将内存中的数据原封不动的进行保存，适用于非字符为主的数据。二进制文件的优点在于存取速度快，占用空间小。本程序采用C语言当中的fscanf函数，可以从文本文件中读取文件。

```
extern int fscanf (FILE *__restrict __stream,const char *__restrict __format,
...) __wur;
```

fscanf函数可以通过传入数据流以及文件的读取格式，以及指定数据流读取去往的地址，来实现文本文件数据的读取。在本程序中，由于需要进行的时候浮点数的计算，所以需要从文件当中读取浮点数，使用的格式为"%f"。类似地，文件的输出可以使用fputs函数来进行将字符串输出到文本当中。由于矩阵当中存储的是浮点数类型，而文本储存的是字符串，所以先要将浮点数通过sprintf函数转化成字符串之后再写入到文件当中。sprintf函数原型如下：

```
/* Write formatted output to S. */
extern int sprintf (char *__restrict __s,const char *__restrict __format, ...)
__THROWNL;
```

2.4 C/C++对比——文件读写

在C++当中，对于文件的读写，只需要将ifstream对象或者ofstream对象与文件关联起来，创建流对象，就可以使用该对象对文件进行操作了，这个对象已经封装好了识别读取类型的方法，只要调用该对象就会匹配你的目标读取或写入类型。

但在C语言当中，文件的读写都要自己去指定类型以及格式，而且在写入文件时还要将源数据流中的数据转化成为字符串才能很好地写入到文件当中，而且C语言采用FILE指针的形式来进行文件的读写，更接近底层。

3.初始化，复制和释放

3.1 初始化矩阵

在本次项目中，一共有三种初始化矩阵的方法。

第一种是读取文本文件对矩阵进行初始化，在初始化矩阵的方法当中，传入了一个char指针，该指针指向文件名。

由于传入的矩阵并没有对内部的浮点数数组分配内存，所以首先需要根据传入的行数以及列数对矩阵内部元素进行初始化。在初始化之前需要对传入数据的合法性进行判断，如果行数和列数不全大于零，由于矩阵的行列数不应该为负数，所以此时不进行初始化，而是提醒输入非法并返回空的矩阵。在分配内存完成之后按照矩阵的行和列依次读入文本文件的浮点数对矩阵内部元素进行初始化。

```
Mat *initMat(Mat *mat, int row, int col) {
  if (row <= 0 || col <= 0) {
    printf("The line or colomn input is improper!\n");
    return NULL;
  }
  // 为矩阵结构各结构体变量赋值
  mat->row = row;
  mat->col = col;
  // 动态分配二维数组的内存
  mat->data = (float *)malloc(sizeof(float) * row * col);
  return mat;
}
```

```
Mat *initMat_file(Mat *mat, int row, int col,
                            char *filename) {
  if (row <= 0 || col <= 0) {
    printf("The line or colomn input is improper!\n");
    return NULL;
  }
  if (filename == NULL) {
    printf("The file name input is improper!\n");
    return NULL;
  }
  // 初始化矩阵
  mat = initMat(mat, row, col);
  // 给矩阵动态分配空间之后，读取相应的数据存储到矩阵中
  if (mat != NULL && mat->row == row && mat->col == col) {
    FILE *fp;
    fp = fopen(filename, "r");
    if (fp == NULL) {
      printf("File open failed!\n");
      exit(1);
      return NULL;
    }
    for (int i = 0; i < row * col; i++)
      fscanf(fp, "%f", &mat->data[i]);
    fclose(fp);
  } else {
    printf("The matrix is not initialized properly!\n");
  }
  return mat;
}
```

第二种是读取float指针对矩阵进行初始化，在初始化矩阵的方法当中，传入了一个float指针，该指针指向矩阵数据。

第三种是用随机数对矩阵进行初始化，在初始化矩阵的方法当中，无需额外传入参数，方便调试。

其实现与第一种方法相似，故只放出方法声明。

```
Mat *initMat_array(Mat *mat, int row, int col, float *data);
Mat *initMat_random(Mat *mat, int row, int col);
```

3.2 复制矩阵

对于一个结构体，复制的方式有两种，一种是直接使用等号进行浅拷贝，该拷贝过程按字节复制的，对于指针型成员变量只复制指针本身，而不复制指针所指向的目标，所以在复制的时候会只将原结构体的指针赋给目标结构体的指针，导致两个结构体的指针指向同一块内存。在本程序中，显然使用浅拷贝无法获得两个相互无关但是内容一样的矩阵结构，所以需要使用深拷贝，对结构体中每个成员逐个赋值。

为了使用深拷贝，且需要避免返回指向被调用函数内部自动变量的指针的问题，事先需要先新创建一个矩阵，之后同时传入新建矩阵以及源矩阵，对源矩阵内数组的每一个元素都逐一复制到新建矩阵中，这样就可以得到两个值完全一样但是内存上毫无关联的矩阵了。若目标矩阵的行数和列数与原矩阵不相同，则释放目标矩阵，重新创建一个行数和列数符合要求的矩阵指针赋值给原来的目标矩阵指针。实现代码如下：

```
bool copyMat(Mat *target, Mat *source) {
  if (source == NULL|| source -> row <= 0 || source -> col <= 0) {
    return false;
  }
  // 如果target矩阵为空，或者target矩阵的行列数与source矩阵不同，则重新初始化target矩阵
  if (target == NULL || source->row != target->row ||
      source->col != target->col) {
    delMat(target);
    Mat *tar = (Mat *)malloc(sizeof(Mat));
    target = initMat(tar, source->row, source->col);
  }
  // 将source矩阵的数据复制到target矩阵中
  for (int i = 0; i < source->row * source->col; i++) {
    target->data[i] = source->data[i];
  }
  return true;
}
```

3.3 释放矩阵

C语言当中局部变量在函数调用结束内存便会被自动释放，与局部变量不同，使用malloc动态分配的空间，它会存储在堆当中，如果没有及时释放，则该内存会一直被占用直到程序结束。在矩阵库实际使用中，储存和使用矩阵的规模可能很大，同时矩阵乘法优化中的Strassen算法也需要为大量的分块矩阵分配内存，用空间换取时间，需要占用的内存很多。如果不及时回收内存，则在该函数调用完毕后会导致大量的内存泄漏，也会严重影响cpu运行效率，对程序本身运行存在极大的负面影响，所以需要及时地进行矩阵内存的回收。

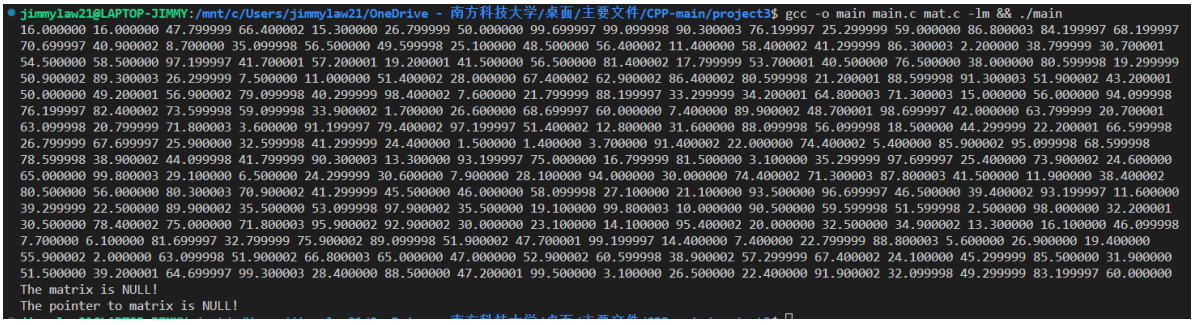
那如何安全地释放矩阵呢？在释放结构体指针指向的内存的时候，即使调用free方法将该结构体的内存释放了，结构体内部的指针指向的内存依然没有被释放。所以对于结构体内部的指针，在释放结构体之前要先进行内部指针指向内存的释放。即使指针指向的内存已经释放，但是此时指针依然指向的是原有内存，但是此时内存已经不可再通过原指针被使用，所以原有的指针应该设置为指向空值NULL。这样子就可以保证矩阵结构已经完全被释放了。

有时候，我们不仅想释放矩阵，还想彻底删除矩阵，减少野指针，因此还设计了一个delete方法，通过传入二级指针来试一级矩阵指针指向null的方法，实现删除矩阵。

具体实现方法如下：

```
// 释放矩阵
bool freeMat(Mat *mat) {
  if (mat == NULL||mat->data==NULL) {
    return false;
  }
  free(mat->data);
  mat->data = NULL;
  free(mat);
  mat = NULL;
  return true;
}

// 删除矩阵
bool delMat(Mat **mat) {
  if (*mat == NULL||(*mat)->data==NULL) {
    return false;
  }
  free((*mat)->data);
  (*mat)->data = NULL;
  free(*mat);
  *mat = NULL;
  return true;
}
// main函数里的测试
Mat *mat1 = (Mat *)malloc(sizeof(Mat));
mat1 = initMat_file(mat1, 16, 16, ch1);
printMat(mat1);
freeMat(mat1);
printMat(mat1);
delMat(&mat1);
printMat(mat1);
```



4.运算与极值

4.1 找矩阵中的最值

本程序实现了两种检查矩阵的方法，分别是寻找矩阵最大值以及最小值，两种方法实现都是通过遍历矩阵内的所有值来寻找最值实现的。由于是直接通过指针进行操作，这两种轻量级的方法在面对大规模输入矩阵的时候避免了大段地复制结构体，同时使用了const关键词，防止指针指向的内存在检查中被篡改，但是由于在typedef定义时未添加const关键词（为了实现释放矩阵方法），所以不能防止指针本

身被篡改。

```
float matMax(const Mat *mat);
float matMin(const Mat *mat);
```

4.2 矩阵与标量运算

本程序实现了矩阵与标量的加减乘除运算，四种方法实现都是通过将运算矩阵的对应位置和标量进行运算之后传入的目标矩阵当中。同最值方法一样，通过指针操作避免了大段复制。

```
bool matAddScalar(Mat *result, const Mat *mat, float scalar);
bool matSubScalar(Mat *result, const Mat *mat, float scalar);
bool matMulScalar(Mat *result, const Mat *mat, float scalar);
bool matDivScalar(Mat *result, const Mat *mat, float scalar);
```

4.3 矩阵加减法

本程序实现了矩阵的加减运算，两种方法实现都是通过将两个运算矩阵的对应位置进行运算之后传入的目标矩阵当中。同最值方法一样，通过指针操作避免了大段复制。

```
bool matAdd(Mat *result, const Mat *mat1, const Mat *mat2);
bool matSub(Mat *result, const Mat *mat1, const Mat *mat2);
```

4.4 矩阵乘法

在本程序中实现了矩阵的乘法运算，实现是通过将两个运算矩阵的对应位置进行运算之后传入的目标矩阵当中。同最值方法一样，通过指针操作避免了大段复制。

由于本次project无需深究性能优化，故只进行访存优化。

首先是朴素矩阵乘法，假设矩阵A的大小为 m × n，矩阵B的大小为n × q，则两个矩阵相乘结果矩阵C的大小为 m× q，两个矩阵相乘的公式为：

$$Ciq = \sum_{i=1}^{n} A_i k B_k q$$

首先是采用朴素的矩阵乘法函数根据上述的公式直观地实现，使用三层循环遍历将两个矩阵相乘得到结果

```
for (int i = 0; i < mat1->row; i++) {
  for (int j = 0; j < mat2->col; j++) {
    for (int k = 0; k < mat1->col; k++) {
      result->data[i * mat2->col + j] +=
          mat1->data[i * mat1->col + k] * mat2->data[k * mat2->col + j];
    }
  }
}
```

然后是访存优化乘法运算，进一步分析，数组是一段连续的内存，无论几维数组都是通过一维的线性内存储存的，所以可以发现在运算的过程当中mat2[k * col + j]在读取内存数据的时候是不连续的，在二维数组当中，mat2[k * col+j]是按照列读取的顺序来读取的，这样会使得其在内存中不断地跳跃，引起访问效率的降低，这种效率的降低当输入的矩阵规模非常大的时候是非常显著的，而且在数据量很大的情况下会消耗巨大的读写内存。因此，要同时实现顺序访问mat1以及mat2, 又要满足下列公式

$$Ciq = \sum_{i=1}^{n} A_i k B_k q$$

，所以可以通过改变循环的顺序来减少内存“跳跃”的次数，实现初步的优化：此时将i的循环放在最外层，将k放在第二层进行循环，将 j的循环放在最内层这样的循环。这样，矩阵乘法运算就实现了访存硬件级优化。完整矩阵乘法代码如下所示：

```
// 矩阵的乘法,访存优化
bool matMul(Mat *result, const Mat *mat1, const Mat *mat2) {
  // 检查矩阵为空
  if (isMatNull(mat1) || isMatNull(mat2)) {
    return false;
  }
  // 检查矩阵的行列数是否相同
  if (mat1->col != mat2->row) {
    printf("The matrix is not the same size!\n");
    return false;
  }
  // 初始化结果矩阵
  if (result == NULL || result->row != mat1->row || result->col != mat2->col) {
    if (!freeMat(result)) {
      printf("Fail freeing matrix");
    }
    Mat *res = (Mat *)malloc(sizeof(Mat));
    result = initMat(res, mat1->row, mat2->col);
  }

  // 矩阵乘法
  for (int i = 0; i < mat1->row; i++) {
    for (int k = 0; k < mat1->row; i++) {
      float temp = mat1->data[i * mat1->col + k];
      for (int j = 0; j < mat1->col; j++) {
        result->data[i * mat1->col + j] += temp * mat2->data[k * mat1->col + j];
      }
    }
  }
  return true;
}
```

4.5 矩阵转置，单位矩阵

本程序实现了矩阵的转置运算，实现方法是通过将原矩阵的对应位置元素传入的目标矩阵的对应位置当中。同最值方法一样，通过指针操作避免了大段复制，使用const关键字防止数据篡改。

```
bool matTranspose(Mat *result, const Mat *mat);
```

同时也实现了单位矩阵生成，为各种矩阵计算提供了便利，实现方法是对传入矩阵的对应位置元素逐一赋值，斜对角线上元素为1，其余为0。在赋值之前，考虑到单位矩阵的特性，除了空指针等的常规检查，还会检查传入矩阵的行数和列数是否相等，以及其是否等于形参整数n。

```
bool getMatUnit(Mat *result, const int n) {
  // 检查矩阵为空
  if (isMatNull(result)) {
```

```
    return false;
  }
  // 检查矩阵是否为方阵
  if (result->row != result->col) {
    printf("The matrix is not square matrix!\n");
    return false;
  }
  // 检查矩阵大小
  if (result->row != n) {
    printf("The matrix is not unit matrix!\n");
    return false;
  }
  // 获得单位矩阵
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (i == j) {
        result->data[i * n + j] = 1;
      } else {
        result->data[i * n + j] = 0;
      }
    }
  }
  return true;
}
```

4.6 矩阵行列式，检查奇异矩阵

本程序实现了矩阵的行列式计算，同最值方法一样，通过指针操作避免了大段复制，使用const关键字防止数据篡改。

该方法会对矩阵首先进行空检查和方阵检查，若矩阵为1，2阶行列式，则使用对角线法进行简单的特化处理，若为3阶以上行列式，则使用代数余子式来计算行列式，期间会生成列数个临时矩阵，在使用完他们后将会对它们进行删除操作，节省内存开销。具体函数如下：

```
// 求矩阵行列式
float matDet(const Mat *mat) {
  // 检查矩阵为空
  if (isMatNull(mat)) {
    return 0;
  }
  // 检查矩阵是否为方阵
  if (mat->row != mat->col) {
    printf("The matrix is not square matrix!\n");
    return 0;
  }
  // 求矩阵行列式
  float det = 0;
  if (mat->row == 1) {  // 1阶行列式
    det = mat->data[0];
  } else if (mat->row == 2) { // 2阶行列式
    det = mat->data[0] * mat->data[3] - mat->data[1] * mat->data[2];
  } else {
    for (int i = 0; i < mat->col; i++) {  // 3阶以上行列式
      Mat *temp = (Mat *)malloc(sizeof(Mat)); // 临时矩阵
      temp = initMat(temp, mat->row - 1, mat->col - 1); // 初始化临时矩阵
```

```
      int k = 0;  // 临时矩阵的行
      for (int m = 1; m < mat->row; m++) {
        for (int n = 0; n < mat->col; n++) {
          if (n != i) {
            temp->data[k] = mat->data[m * mat->col + n];
            k++;
          }
        }
      }
      det += pow(-1, i) * mat->data[i] * matDet(temp);  //代数余子式计算行列式
      delMat(&temp);    //删除临时矩阵
    }
  }
  return det;
}
```

本程序实现了奇异矩阵检查，在函数里分别检查矩阵为空，矩阵为非方阵，行列式是否为0三种错误情况。

```
bool isMatSingular(const Mat *mat);
```

4.7 矩阵求逆

本程序实现了矩阵的求逆运算，实现方法是通过将原矩阵的对应位置元素经过伴随阵法运算后传入的目标矩阵的对应位置当中。同最值方法一样，通过指针操作避免了大段复制，使用const关键字防止数据篡改。

关于伴随矩阵法：

定理：n阶矩阵$A = [a_{ij}]$为可逆的充分必要条件是A非奇异，且：

$$A^{-1} = \frac{1}{|A|}\begin{bmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{bmatrix}$$

其中，A_{ij}是|A|中元素a_{ij}的代数余子式；矩阵

$$\begin{bmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{bmatrix}$$

称为矩阵A的伴随矩阵，记作A*，于是有

$$A^{-1} = \frac{1}{|A|}A^{*}。$$

由伴随矩阵法，我们可以根据上述的公式直观地实现，使用三层循环得到结果，期间会生成列数个临时矩阵，在使用完他们后将会对它们进行删除操作，节省内存开销。具体函数如下：

```
bool matInverse(Mat *result, const Mat *mat) {
  // 检查矩阵为空
  if (isMatNull(mat)) {
    return false;
  }
  // 检查矩阵是否为方阵
  if (mat->row != mat->col) {
    printf("The matrix is not square matrix!\n");
    return false;
  }
  // 检查矩阵是否为奇异矩阵
  if (isMatSingular(mat)) {
    printf("The matrix is singular matrix!\n");
    return false;
  }
  // 初始化结果矩阵
  result = initMat(result, mat->row, mat->col);
  // 矩阵求逆
  float det = matDet(mat);
  for (int i = 0; i < mat->row; i++) {
    for (int j = 0; j < mat->col; j++) {
      Mat *temp = (Mat *)malloc(sizeof(Mat));
      temp = initMat(temp, mat->row - 1, mat->col - 1);
      int k = 0;
      for (int m = 0; m < mat->row; m++) {
        for (int n = 0; n < mat->col; n++) {
          if (m != i && n != j) {
            temp->data[k] = mat->data[m * mat->col + n];
            k++;
          }
        }
      }
      result->data[j * mat->row + i] = pow(-1, i + j) * matDet(temp) / det;
      delMat(&temp);
    }
  }
  return true;
}
```

5.其他方法

1.随机生成矩阵数据&打印矩阵

本程序准备了三种方法来随机生成数据，分别是给已有矩阵随机赋值，生成随机矩阵数据文件或矩阵数据指针，以及一个二维打印矩阵的方法，方便进行程序调试。

```
void random_mat(Mat *mat); // 给矩阵随机赋值

char *random_mat_file(int row, int col, char *filename); // 随机生成矩阵数据文件

float *random_mat_ptr(int row, int col); // 随机生成矩阵数据指针

void printMat(const Mat *mat);
```

2.违法矩阵检查

本程序在进行大部分的矩阵操作之前都要对传入的矩阵进行检查，而检查矩阵指针为空，矩阵数据指针为空，矩阵行列数非零则是最常见的检查方面，因此将它们提取出来放进一个检查方法里以备调用，减少了大量冗余代码，使得代码的美观程度和可维护性得到提升。

```
// 检查矩阵是否为空
bool isMatNull(const Mat *mat) {
  if (mat == NULL) {
    printf("The pointer to matrix is NULL!\n");
    return true;
  }
  if (mat->data == NULL) {
    printf("The pointer to matrix data is NULL!\n");
    return true;
  }
  if (mat->row <= 0 || mat->col <= 0) {
    printf("The matrix is NULL!\n");
    return true;
  }
  return false;
}
```

6.程序的健壮性

1.文件读写的健壮性

本程序使用的是fopen方法进行文件的读取，fopen方法在文件读取成功的时候会返回该文件对应的FILE类型指针，如果打开失败则返回NULL。在用文件名初始化矩阵方法中，会对打开文件之后返回的FILE值进行判断，根据返回值是否为空来判断文件是否成功打开。如果该指针是空值，则会报告文件打开异常之后便会提示错误信息并退出程序。这一操作保证了程序中的矩阵始终正常地存取了文本文件当中的浮点数，否则对于文件的操作将会是基于一个空指针来操作，极其容易导致程序崩溃。

2.初始化的健壮性

在矩阵初始化的过程中，调用矩阵初始化方法并通过传入矩阵的行列数对矩阵本身以及内部元素进行初始化。但是如果输入的行数或者列数小于等于零则会导致矩阵不能正常初始化，此时该函数将提示错误信息并返回一个空值。

3.操作函数的健壮性

分为两个方面：

在复制矩阵或者是进行矩阵乘法运算的时候，容易出现输出的目标矩阵为空的情况，或者是该矩阵的大小与源矩阵不一致。因此，在本程序中，若出现上述情况，首先将会通过指向原矩阵的指针将原内存块进行释放，之后重新初始化一个新的矩阵，将新矩阵的规模设置为与源矩阵一致，在将原指针指向这一块新的内存块，实现矩阵的纠正。这样保证了矩阵操作和矩阵运算的合理性与可行性，使得程序更加健壮。

在其余情况，也容易出现输出的目标矩阵或数据指针为空的情况，或者是该矩阵的大小与源矩阵不一致，所以在操作开始前都会通过isMatNull函数检查，然后提示错误信息并返回false或者空值。

Part 3 - Result & Verification

1.三种方法生成浮点数矩阵

```
【Testcase#1】
  Mat *mat1 = (Mat *)malloc(sizeof(Mat));
  Mat *mat2 = (Mat *)malloc(sizeof(Mat));

  mat1 = initMat_file(mat1, 4, 4, ch1);
  mat2 = initMat_file(mat2, 4, 4, ch2);

  Mat *mat3 = (Mat *)malloc(sizeof(Mat));
  Mat *mat4 = (Mat *)malloc(sizeof(Mat));

  mat3 = initMat_array(mat3, 4, 4, mata);
  mat4 = initMat_array(mat4, 4, 4, matb);

  Mat *mat5 = (Mat *)malloc(sizeof(Mat));
  Mat *mat6 = (Mat *)malloc(sizeof(Mat));

  mat5 = initMat_random(mat5, 4, 4);
  mat6 = initMat_random(mat6, 4, 4);
```

```
4.000000 4.000000 47.799999 66.400002
15.300000 26.799999 50.000000 99.699997
99.099998 90.300003 76.199997 25.299999
59.000000 86.800003 84.199997 68.199997

4.000000 4.000000 8.700000 35.099998
56.500000 49.599998 25.100000 48.500000
56.400002 11.400000 58.400002 41.299999
86.300003 2.200000 38.799999 30.700001

97.199997 41.700001 57.200001 19.200001
41.500000 56.500000 81.400002 17.799999
53.700001 40.500000 76.500000 38.000000
80.599998 19.299999 50.900002 89.300003

26.299999 7.500000 11.000000 51.400002
28.000000 67.400002 62.900002 86.400002
80.599998 21.200001 88.599998 91.300003
51.900002 43.200001 50.000000 49.200001

56.900002 79.099998 40.299999 98.400002
7.600000 21.799999 88.199997 33.299999
34.200001 64.800003 71.300003 15.000000
56.000000 94.099998 76.199997 82.400002

73.599998 59.099998 33.900002 1.700000
26.600000 68.699997 60.000000 7.400000
89.900002 48.700001 98.699997 42.000000
63.799999 20.700001 63.099998 20.799999
```

2.复制，释放和删除矩阵

```
【Testcase#2】
if (cpyMat(mat5, mat6)) {
    printf("Copy successfully\n");
    printMat(mat5);
    printf("\n");
    printMat(mat6);
} else {
    printf("Copy failed\n");
}
if (delMat(&mat5)) {
    printf("Delete successfully\n");
  } else {
    printf("Delete failed\n");
  }
  printMat(mat5);
  if (freeMat(mat6)) {
    printf("Free successfully\n");
  } else {
    printf("Free failed\n");
  }
  printMat(mat6);
```

```
Copy successfully
73.599998 59.099998 33.900002 1.700000
26.600000 68.699997 60.000000 7.400000
89.900002 48.700001 98.699997 42.000000
63.799999 20.700001 63.099998 20.799999

73.599998 59.099998 33.900002 1.700000
26.600000 68.699997 60.000000 7.400000
89.900002 48.700001 98.699997 42.000000
63.799999 20.700001 63.099998 20.799999
Delete successfully
The pointer to matrix is NULL!
Free successfully
The matrix is NULL!
```

3.矩阵求极值

【Testcase#3】

```
printf("Max of mat1 is %f\n", matMax(mat1));
printf("Min of mat1 is %f\n", matMin(mat1));
```

mat1文件：

```
47.800000 66.400000 15.300000 26.800000 50.000000 99.700000 99.100000 90.300000
76.200000 25.300000 59.000000 86.800000 84.200000 68.200000 70.700000 40.900000
```

```
Max of mat1 is 99.699997
Min of mat1 is 15.300000
```

4.矩阵标量计算

【Testcase#4】

```
  Mat *res = (Mat *)malloc(sizeof(Mat));
  res = initMat_random(res, 4, 4);
  printMat(res);
  matAddScalar(res,mat1, 1.0f);
  printMat(res);
  matSubScalar(res, mat1, 1.0f);
  printMat(res);
  matMulScalar(res, mat1, 2.0f);
  printMat(res);
  matDivScalar(res, mat1, 2.0f);
  printMat(res);
```

```
71.800003 3.600000 91.199997 79.400002
97.199997 51.400002 12.800000 31.600000
88.099998 56.099998 18.500000 44.299999
22.200001 66.599998 26.799999 67.699997

48.799999 67.400002 16.299999 27.799999
51.000000 100.699997 100.099998 91.300003
77.199997 26.299999 60.000000 87.800003
85.199997 69.199997 71.699997 41.900002

46.799999 65.400002 14.300000 25.799999
49.000000 98.699997 98.099998 89.300003
75.199997 24.299999 58.000000 85.800003
83.199997 67.199997 69.699997 39.900002

95.599998 132.800003 30.600000 53.599998
100.000000 199.399994 198.199997 180.600006
152.399994 50.599998 118.000000 173.600006
168.399994 136.399994 141.399994 81.800003

23.900000 33.200001 7.650000 13.400000
25.000000 49.849998 49.549999 45.150002
38.099998 12.650000 29.500000 43.400002
42.099998 34.099998 35.349998 20.450001
```

5.访存硬件级优化对矩阵乘法速度对比

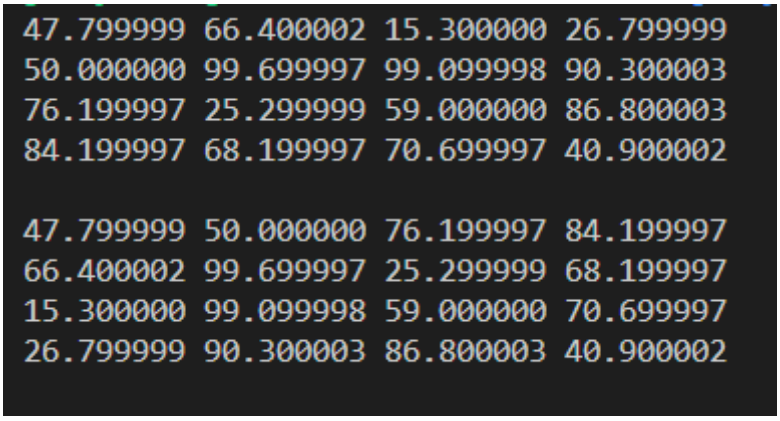
```
【Testcase#5】
 Mat *res = (Mat *)malloc(sizeof(Mat));
 res = initMat_random(res, 1024, 1024);
 // printMat(mat1);
 // printMat(mat2);
 struct timeval start1,end1;
 struct timeval start2,end2;

 gettimeofday(&start1, NULL);
 matMul(res, mat1, mat2);
 gettimeofday(&end1, NULL);
 printf("OPT Time cost: %ld us\n", (end1.tv_sec - start1.tv_sec) * 1000000 +
                                   (end1.tv_usec - start1.tv_usec));
 // printMat(res);
 gettimeofday(&start2, NULL);
 matMulNaive(res, mat1, mat2);
 gettimeofday(&end2, NULL);
 printf("Naive Time cost: %ld us\n", (end2.tv_sec - start2.tv_sec) * 1000000 +
                                   (end2.tv_usec - start2.tv_usec));
 //  printMat(res);
```

```
OPT Time cost: 3506493 us
Naive Time cost: 4890237 us
```

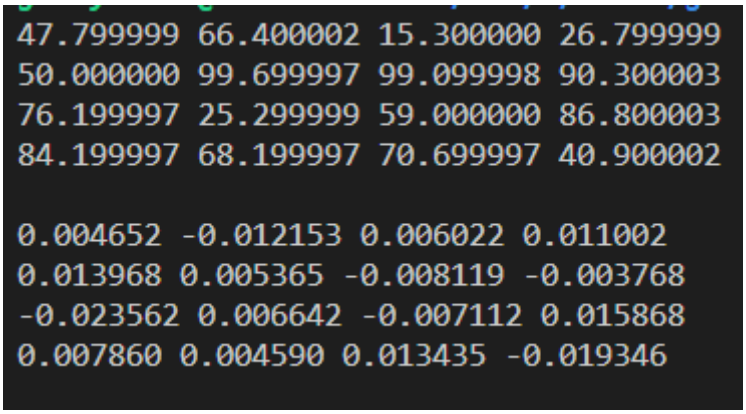
6.矩阵求行列式

```
【Testcase#6】
 Mat *res = (Mat *)malloc(sizeof(Mat));
 res = initMat_random(res, 4, 4);
 printMat(mat1);
 matTranspose(res, mat1);
 printMat(res);
```



7.矩阵求逆

```
【Testcase#7】
 Mat *res = (Mat *)malloc(sizeof(Mat));
 res = initMat_random(res, 4, 4);
 printMat(mat1);
 matInverse(res, mat1);
 printMat(res);
```



Part 4 - Difficulties&Solutions

1.函数调用复制结构体速度慢问题及解决

在矩阵规模非常大的时候，矩阵操作的所需时间非常长，结合课上所学内容，知道了原因是调用函数的时候是按值传递，会复制一遍传入的参数，这样会严重拖慢程序的速度，改进之后将传入结构体本身改为传入指向该结构体内存的指针，而指针只是一个32位整数，这样就避免了复制矩阵结构体的问题，显著地加快了程序的运行速度。

2.函数返回指向自动变量内存的指针问题及解决

在对矩阵进行操作的时候，一开始在被调用函数当中创建了自动变量来存储运算值，而在返回的时候直接将指向该自动变量内存的指针进行了返回，由于自动变量指向的内存在函数执行完毕之后就会被释放，所以返回的实际上是指向一块空内存空间的指针，在使用该指针的时候就会导致系统崩溃。

后来通过提前创建目标矩阵并为其动态分配内存空间，之后再作为函数实参传入被调用函数，在被调用函数里面给目标矩阵内元素进行赋值，最后返回指向该目标函数内存块的指针，这样就有效避免了函数返回自动变量指向内存指针的问题。

3.无法在函数内删除矩阵指针问题及解决

一开始设计释放矩阵方法时，已经做到了在释放矩阵前释放矩阵内的数据指针，并且在数据指针和矩阵指针指向的内存释放完后，将它们各自指向空值。后来，在与同学们讨论过后发现该指针自身仍存在于内存当中，为了更加彻底地释放内存，又设计了删除矩阵地方法，通过传入矩阵的二级指针来进行释放内存和指针指空操作，最后通过访问指针的地址成功删除了矩阵指针，有效解决了无法在函数内删除矩阵指针的问题。

4.无法编译内联函数的问题及解决

在将一些小函数设置为内联函数后，我一开始无法正常编译文件，遇到了如下报错信息warning: inline function xxx declared but never defined，后来经过查询资料，发现原因在于gcc编译器对于inline函数处理的方式不同导致。目前C语言有几个国际标准 C89 C99 C11，不同标准中对于inline函数的处理不一样的，如果gcc编译产生如上问题，可能是由于对于inline函数的解释标准过高，需要使用传统模式进行编译。因此，在采用-fgnu89-inline编译选项后，成功解决了这个问题。

Part 5 - Conclusion

这次project是我第一次使用C语言编写程序，通过这次使用C语言设计矩阵结构体，我除了对C语言更加熟悉之外，也一直在关注C语言与C++不同的地方，对这两门编程语言的不同点以及相同点有了更深刻的认知。同时感受到C语言更加贴近底层，如动态分配时还需要指定分配的字节数目，在使用文件指针的时候也许多在C++当中封装起来的方法，都依赖于编写者书写正确的类型。

而在结构体设计上，这也是我第一次设计一个结构体和其操作函数，之前一直是使用java和c++设计类，通过这次使用C语言设计矩阵结构体，我不仅通过使用指针和箭头运算符来访问内部元素，更加熟练了结构体的操作，还对结构体和类的相同点和不同点有了更深的认识。

同时，我还对底层内存实现有了更深的理解：通过使用一维数组来模拟二维数组的存储，经过一维数组进行二维数组内部元素的访问是要通过在行位置和列位置计算才能取到目标的数据，这样的操作也让我对数组的存储方式有了更深的理解。

由于C语言没有引用，起初通过直接传值的方式会复制大量的数据从而拖慢程序的运行，后来通过直接传递指针的方式传递地址，直接对指向的内存进行操作，综合使用数组以及结构体也更加让我了解了指针的本质以及相关操作。同时，通过使用指针进行内存释放，释放内存后将指针设为空和使用二级指针删除指针，我对内存管理和程序健壮性有了更深的认识。

当然，这次project也有很多不足之处，譬如没有实现更多的操作函数，如求特征值，以及没有实现更好的指针管理，如指针表，但是我会进一步学习相关内容，尽可能在下一次project中用上相关技术。

以上就是这次报告的全部内容，感谢老师的阅读！