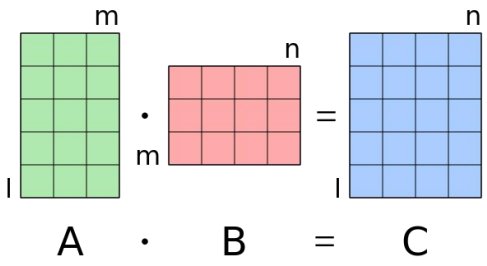
CS205 C/C++ Program Design – Project 1.

Please implement a function to compute the multiplication of two matrices. The data type of matrix elements is `float` (not `double`). As shown in the following figure[1], the resulting matrix **C**, known as the matrix product, has the number of rows of the first matrix **A** and the number of columns of the second matrix **B**.



The definition of matrix multiplication is as follows[2]:

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

Requirements:

1. (20 points) Please define a `Struct` for matrices. The Struct should contain the property descriptions such as the number of rows, the number of columns, data, etc.
2. (30 points) Please implement a function which can compute the matrix product of two matrices. If the sizes of the two input matrices do not match, the function should report errors. The function must be robust and cannot crash when you input something

[1] https://en.wikipedia.org/wiki/File:Matrix_multiplication_qtll.svg
[2] https://en.wikipedia.org/wiki/Matrix_multiplication

incorrectly.

3. (10 points) Please measure how many milliseconds (or seconds) for the multiplication when the two matrices contain about 200M elements. **Hint**: You can follow the example at https://en.cppreference.com/w/cpp/chrono/time_point to measure the time.
4. (25 points) Improve the efficiency of your source code. Please report your methods and the improvements. **Deep analysis and comparisons are expected in the report.**
5. (10 points) Compare your implementation with OpenBLAS https://github.com/xianyi/OpenBLAS Please report the differences on results and speed. **Hint**: function `cblas_sgemm()` in OpenBLAS.
6. (5 points) Please host your source code at GitHub.com. you can just put a link in the report. If you do not host your source code at GitHub.com, please upload your source with your report to Blackboard.
7. Your total score will also be affected by your source code quality and report quality.

Rules:

1. Please submit your assignment report before its deadline. After the deadline (even 1 second), 0 score!
2. Do not code your program unnecessarily complex. Simple is beautiful!
3. 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

Midterm Project: Matrix Multiplication

Name: 王奕童(YeeTone Wang)

SID: 11910104

Course Instructor: 于仕琪(Shiqi Yu)

【说明】本次期中 project 融合了 Assignment2（计算器）和 Assignment3（向量点积）的部分功能，并在用户体验和运行效率上有适当的改进和提高。本次报告共 41 页，包含大量截图和说明，阅读需要一定的时间。

Part 1. Description

本次期中 Project 一共上交 6 个代码文件：

1. main.cpp -->计算器主程序
2. test.cpp -->用于文档中各个样例的测试文件
3. CMatrix.h --> C++语言实现的矩阵类头文件
4. CMatrix.cpp --> C++语言实现的矩阵类头文件
5. Main.java --> Java 语言的测试主程序
6. JavaMatrix.java -->Java 语言实现的矩阵类

【最初版本运行时间】两个 1024*1024 的密集随机方阵相乘用时：13000ms~14000ms

【最终版本运行时间】两个 1024*1024 的密集随机方阵相乘用时：80ms~120ms

【特殊】在应对特殊类型矩阵（如零矩阵，单位阵，对角阵，稀疏阵）的乘法时，效率有显著提高。

【基本要求实现】

1. 矩阵类的定义与实现（Requirement1）
2. 矩阵乘法函数实现（Requirement2）
3. 矩阵相乘时行列不匹配的情形的检测（Requirement2）
4. 用户输入错误时提示用户重新输入（Requirement2）
5. 阻止用户输入过大的矩阵行/列数值，避免内存超限（Requirement2）
6. 修正矩阵运算时 float 类型精度丢失问题

【时间测量方法】

1. 毫秒时间测量方法介绍【本次报告时间测评方法】
2. 纳秒时间测量方法介绍【高精度时间测评方法】

【效率提升方法】

0. 第一个版本的运行时间说明

1. 函数值传递改为指针传递（Requirement4）
2. 简单函数 inline 处理（Requirement4）
3. 改进 C/C++的 IO 效率（Requirement4）
4. O3 优化编译选项（Requirement4）
5. 多线程并行计算（Requirement4）
6. 使用指令集加速计算（Requirement4）
7. 提升随机数的生成速度（Requirement4）
8. 数组存储结果再合并（Requirement4）
9. 空间使用与回收（Requirement4）
10. 编译模式由 Debug 模式转为 Release 模式（Requirement4）
11. 更换矩阵运算的循环次序（Requirement4）
12. 根据矩阵的特殊性决定运算策略（Requirement4）

【200M大矩阵的运算计时（与OpenBLAS比较）】

1. 密集阵*密集阵（Requirement3，5）
2. 密集阵*稀疏阵（Requirement3，5）
3. 密集阵*对角阵（Requirement3，5）
4. 向量*向量（Requirement3，5）
5. 密集阵*单位阵（Requirement3，5）
6. 密集阵*零矩阵（Requirement3，5）

【与Java语言的比较与分析】

1. 关于Java代码的简要说明
2. Java/C++两种语言的测试效果展示
3. 具体的原因与分析

【提升用户体验】

1. 支持对用户显示欢迎信息
2. 支持对用户显示帮助信息
3. 支持用户选择随机数测试模式
4. 支持对用户显示错误信息并提示用户重新输入
5. 支持用户自由切换中英双语
6. 支持用户自由切换矩阵显示模式
7. 支持用户自由随时退出
8. 支持用户在计算完毕后对计算器重新利用而不是直接退出

【额外功能实现】

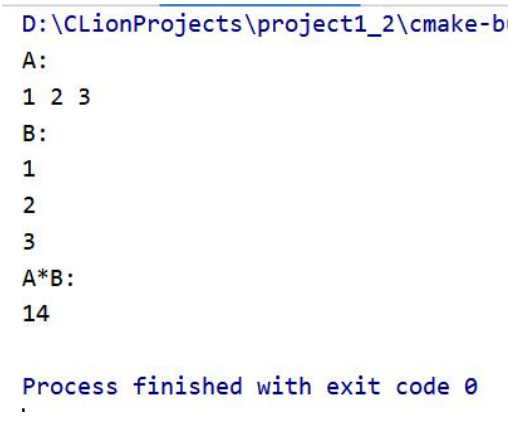
1. 实现根据阶数生成单位矩阵
2. 实现根据大小生成零矩阵
3. 实现矩阵加/减法
4. 实现矩阵转置
5. 实现矩阵的对角元素赋值为统一值的对角阵
6. 实现矩阵内部元素的批量填充
7. 实现允许根据下标获取/修改矩阵内部的值
8. 实现判断两矩阵是否相等
9. 实现方块矩阵的快速幂

【说明】本次作业报告的测评环境：
运行系统：Windows10
编译器：mingw-GCC
内存限制：2048M
C++语言版本：C++20
CPU核数：8
运行IDE：CLion 2020.2
运行机器：联想小新Air-14IIIL 2020

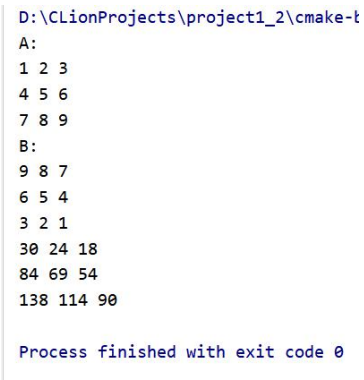
Part 2. Result & Verification

In this part, you should present the result of your program by listing the output of test cases and optionally add a screen-shot of the result.

Test case #1:
请在test.cpp中调用此函数以测试
```
void test_case1();
```
Input:
Matrix1：(1*3)
$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$
Matrix2：(3*1)
$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
Output: $\begin{bmatrix} 14 \end{bmatrix}$
Screen-short for case #1:



Test case #2:
请在test.cpp中调用此函数以测试
```
void test_case2();
```
Input:
Matrix1：(3*3)
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
Matrix2：(3*3)
$\begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$
Output: $\begin{bmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{bmatrix}$
Screen-short for case #2:



Part 3. Difficulties & Solutions, or others

【基本要求实现】

1.矩阵类的定义与实现（Requirement1）

【类的具体代码实现】class CMatrix;
【类的内部整体架构】请见以下3张截图

```
class CMatrix {
private:
    bool isSquare= false;
    bool isDiagonal= false;
    bool isIdentical=false;
    bool isZeroMatrix= false;
    int nonZeroNumberNotMain=0;
    int nonZeroNumberMain=0;
    int nonOneNumberMain=0;
    int nonZeroNumber=0;
private:
    int row=0;
    int column=0;
    float* data= nullptr;
```

```
public:
    static bool isExpressedOmitted;
    explicit CMatrix(int r,int c);
    [[nodiscard]] float get(int r,int c) const;
    void set(int r,int c,float value);
    void printCM() const;
    void fill(float value);
    void inverseSign();
    void setDiagonal(float value);
    [[nodiscard]] CMatrix transport()const;
    static CMatrix getIdentical(int x);
    static inline CMatrix getZeros(int r,int c){...}
    inline CMatrix operator*(const CMatrix& cm)const{...};
    inline CMatrix operator+(const CMatrix& cm)const{...};
    inline CMatrix operator-(CMatrix& cm)const{...};
    inline CMatrix operator^(int x){...}
    inline bool operator==(const CMatrix& cm)const{...}
```

```
private:
    static CMatrix subMultiply(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float* result
    static void subAdd(const CMatrix* cm1,const CMatrix* cm2,int start,int end);
    static CMatrix vectorMultiply(const CMatrix* vector1,const CMatrix* vector2);
    static void subVectorMultiply(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float* res
    static inline bool check2MatrixMultiplyMatch(const CMatrix* cm1,const CMatrix* cm2){...}
    static inline CMatrix zeroMatrixMultiplication(const CMatrix* cm1,const CMatrix* cm2){...}
    static CMatrix diagonalMatrixMultiplication(const CMatrix* cm1,const CMatrix* cm2);
    static CMatrix identicalMatrixMultiplication(const CMatrix* cm1,const CMatrix* cm2){...}
    CMatrix singleMultiply(const CMatrix* cm) const;
    CMatrix multipleMultiply(const CMatrix* cm) const;
    CMatrix addMatrix(const CMatrix* cm) const;

};
```

【内部成员变量】
①数据成员变量:

```
private:
    int row=0;//描述矩阵的行数
    int column=0;//描述矩阵的列数
    float* data= nullptr;//表示用一维数组表示矩阵内部元素
```

②矩阵类型描述成员变量:

```
private:
    bool isSquare= false;//描述矩阵是否是方阵
    bool isDiagonal= false;//描述矩阵是否是对角阵
    bool isIdentical=false;//描述矩阵是否是单位阵
    bool isZeroMatrix= false;//描述矩阵是否是零矩阵
    int nonZeroNumberNotMain=0;//描述除主对角线其他位置的非零元的个数
    int nonZeroNumberMain=0;//描述主对角线上非零元的个数
```

```
    int nonOneNumberMain=0;//描述主对角线上非1元素的个数
    int nonZeroNumber=0;//描述全矩阵里面非零元的个数
```

③矩阵表示方法描述成员变量：

```
public:
    static bool isExpressedOmitted;//描述矩阵打印时是否省略部分元素
```

【关于利用一维数组存储二维数组】

该类的存储方法为按行优先存储，如下图所示：

2.矩阵乘法函数实现（Requirement2）

【矩阵乘法运算规则】请见本次报告的第一页。

【设计思路】先令用户分次输入矩阵大小存入全局变量m，n，k，然后立即利用new关键字对全局的矩阵指针变量cmPointer1和cmPointer2进行开辟空间初始化，分别指向两个CMatrix变量：

```
cmPointer1=new CMatrix(m,n);
cmPointer2=new CMatrix(n,k);
```

然后再建一个CMatrix指针的全局变量cmResultPointer来指向结果即可。

运算过程中，先根据自身矩阵以及另一矩阵的特性来决定使用什么样的运算方法。

①密集/稀疏情形：本矩阵行数多于CPU核数则采用多线程+指令集的并发计算方法；否则采用单线程的计算方法；

②零矩阵情形：采用直接生成相应默认矩阵的方法即可；

③单位阵情形：采用直接返回另一矩阵的方法即可；

④对角阵情形：采用对于对角线元素遍历乘以另一矩阵的行/列即可；

⑤向量情形：采用Assignment3的优化计算方法即可。

【函数实现】为了在封装多个函数判断下能够使得乘法简单易懂，在此使用了重载运算符“*”来实现矩阵乘法：

文件：CMatrix.h

运算符重载：`inline CMatrix operator*(const CMatrix& cm)const;`

【难易系数】☆☆☆☆

【难点分析】循环逻辑的判断以及循环次序的调整，多种特殊类型矩阵的考虑以及乘法实现以降低时间复杂度

Test case #3:

请在test.cpp中调用此函数以测试

```
void test_case3();
```

$$\begin{vmatrix}1&2&3&4&5\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{vmatrix} * \begin{vmatrix}1&0&0&0&0\\2&0&0&0&0\\3&0&0&0&0\\4&0&0&0&0\\5&0&0&0&0\end{vmatrix} = \begin{vmatrix}55&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{vmatrix}$$

Screen-short for case #3:

```
A:
1 2 3 4 5
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
B:
1 0 0 0 0
2 0 0 0 0
3 0 0 0 0
4 0 0 0 0
5 0 0 0 0
A*B:
55 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Process finished with exit code 0
```

3.矩阵相乘时行列不匹配的情形的检测（Requirement2）

【合理性判断规则】根据线性代数的计算法则，AB两个矩阵能够相乘则必须满则A的列数等于B的行数才能相乘。
【设计思路】直接建立一个函数判断两矩阵的行列描述元素是否匹配即可。如匹配则继续运算，否则显示错误信息并返回空矩阵。
【函数实现】

```
private:
static inline bool check2MatrixMultiplyMatch
(const CMatrix* cm1,const CMatrix* cm2);
```

【难易系数】 
【难点分析】无显著难点，只需要判断好到底是两个元素相等即可。

Test case #4: 行列大小不匹配
请在test.cpp中调用此函数以测试

```
void test_case4();
```

$$\begin{bmatrix}0 & 0 & 0\end{bmatrix} * \begin{bmatrix}0\\0\end{bmatrix} = ?$$

Screen-short for case #4:

```
D:\CLionProjects\project1_2\cmake-build-release\project_test.exe
These 2 Matrix Size is not Matched! No multiplication is allowed!

Process finished with exit code 0
```

4.用户输入错误时提示用户重新输入（Requirement2）

【检测规则】检测用户输入的内容不符合要求的情形，如：输入矩阵的大小描述元素m,n,k时，输入浮点数、负数或者其他字符；输入矩阵元素时包含其他非浮点数字符(如*&^%$#等)

【设计思路】建立`while(true)`的无限循环，先输入string进行存储，然后在`try-catch`中使用stoi函数。如果stoi函数正常，则退出循环，函数体结束；如果输入输入矩阵的大小描述元素时为负数或者超过30000（避免内存超限异常退出），则可以选择抛出`out_of_range`异常；如果输入异常字符则`stoi`函数自动会抛出`invalid_argument`异常。然后可以根据`catch`语句块捕获的异常来相应调用`cerr`来输出相应的错误信息，并让用户继续循环输入。

【函数实现】

```
void cinSize(char sign);
inline void cinAllElement(int sign);
inline void cinElement(int r,int c,int sign);
```

【难易系数】 

【难点分析】无显著难点，只需要学习stoi函数的异常处理以及C++类库里的exception类的一些函数如`what()`等等即可。

【参考链接】`stoi`函数可能出现的异常：
https://blog.csdn.net/u014694994/article/details/79074566/

【参考链接】C++的异常处理：
https://www.runoob.com/cplusplus/cpp-exceptions-handling.html

Test case #5: 用户输入错误
请在main.cpp中自行输入内容以测试
试图让A的矩阵尺寸为23*23q
然后A[0][0]=26u
Screen-short for case #5:

```
Please input for m
23
Please input for n
23q
Please input for n
You have input something wrong!
If you want to make matrix2 be a
q
1781120459 2041300628 1688501987
Please input for matrix1[0][0]
23u
You have input something wrong!
```

5.阻止用户输入过大的行与列（Requirement2）

【检测规则】在先前的 Assignment3（向量点积）中，我们已经发现本机能够存储容纳的最大向量长度为 99999999。因此，我们计算得出最大可行的 float 内存容纳为

4*2*99999999 个字节，因此我们考虑最坏情形：输入+输出共 3 个矩阵，均为方阵且方阵的行数与列数均为最大值，因此可以容纳的最大行数和列数为：

$\sqrt{\frac{2*999999999}{3}}=25819$。在这里考虑到其他因素，我们规定最大的行列数为 25000。

【设计思路】在先前提到的 `stoi` 函数中，如果转出的数字大于 25000，则我们主动抛出 `out_of_range` 异常，以交给 `catch` 语句块进行处理。

【函数实现】此函数中的`try-catch`语句块中有相应的判断与实现。

```
void cinSize(char sign);
```

【难易系数】⭐⭐

【难点分析】需要对本机电脑的内存有一定的估计；需要考虑到 C++程序在运行当中对于系统资源的使用情况；需要对于 C++的异常类有一定的了解。

Test case #6: 用户输入过大矩阵描述元素

请在 main.cpp 中自行输入内容以测试

试图让A的矩阵尺寸为m=25000，n=25001

```
Please enjoy your matrix calculation!
Please input for m
25000
Please input for n
25001
Please input for n
The number is too large! Out of Memory!
```

6.修正矩阵运算时float类型精度丢失的问题

（在 Assignment3 的报告中已经有相应的分析）

【问题概述】**#Test case7:**

考虑向量情形：（1*200M）*（200M*1）

$$\begin{bmatrix}1 & 1 & \dots & 1 & 1\end{bmatrix}*\begin{bmatrix}1\\1\\\dots\\1\\1\end{bmatrix}=?$$

根据线性代数的乘法法则，理论上：$[2e+8]$

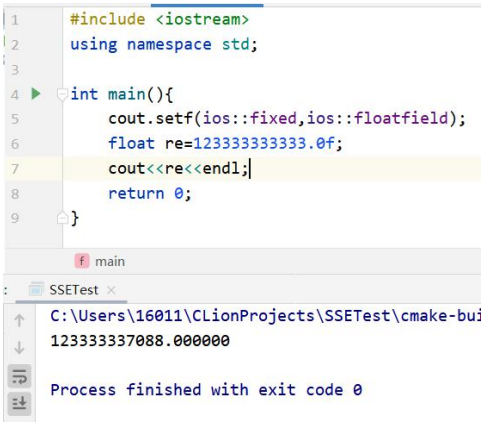
但是实际上测试结果为：

```
1.67772e+08

Process finished with exit code 0
```

，出现较大误差。

【原因分析】这是源于float类型的存储精度有限。
float类型：4bytes，6-7位有效数字，数值范围：1.4E-45~3.4E+38
double类型：8bytes，15-16位有效数字，数值范围：4.9E-324~1.7E+308
在float的数字超过精度范围时，数值就会发生错误，如



【难易系数】 
【难点分析】需要深刻理解float类型与double类型的差异，以及float与double之间运算精确度的规则。
【解决方案】改用double类型存储运算结果，即可回到正确运算结果。（运算过程中，两个float元素先提升类型为double，然后再进行运算赋给double变量，最后才使用(float)来强行转换为float类型，尽可能地减小运算结果的损失。
【修复后】

```
2e+08

Process finished with exit code 0
```

【参考链接】https://blog.csdn.net/a10201516595/article/details/10333034

【时间测量方法】（同Assignment3）

1. 毫秒时间测量方法介绍【本次报告时间测评方法】

【头文件要求】

```
#include <ctime>
```

【对应函数】
clock()，返回类型为long
【使用方法】
开始前用一个long变量t1存储开始时间，结束时用另一个long变量t2存储结束时间，t2-t1即为运行所用的毫秒时间
【参考链接】https://blog.csdn.net/qq_26836575/article/details/78488358

2. 纳秒时间测量方法2介绍【高精度时间测评方法】

【头文件要求】

```
#include <ctime>
```

【对应函数】

`chrono::steady_clock::now().time_since_epoch().count()`

返回类型为`long long`

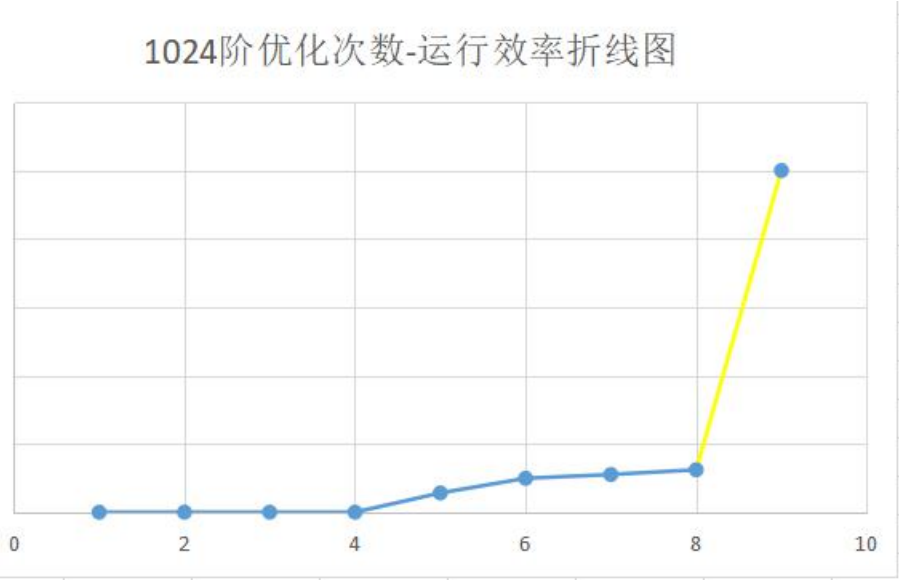
【使用方法】

开始前用一个`long long`变量`t1`存储开始时间，结束时用另一个`long long`变量`t2`存储结束时间，`t2-t1`即为运行所用的纳秒时间

【参考链接】https://blog.csdn.net/qq_31175231/article/details/77923212

【效率提升方法】

优化次数-运行效率的折线图如下所示：



（蓝色折线代表效率提升对于任何类型的矩阵都有效，黄色折线代表效率提升仅对特殊类型矩阵有效）

0.第一个版本的运行时间说明

【原始效果】

测试用例：两个1024*1024阶矩阵相乘

随机数生成时间：60ms-80ms

矩阵乘法计算时间：13000ms-14000ms

详情请见以下的截图：

```
D:\CLionProjects\cppPro1\cmake-build-debug\cppPro1.exe
Data IO Time:63ms
Start!
13398ms

Process finished with exit code 0
```

1.函数值传递改为指针传递（Requirement4）

【问题分析】第一版的计算主函数：

```
public:
    Matrix multiply(Matrix matrix2);
```

该函数为传递Matrix对象的值传递，并且返回一个Matrix对象。在这个过程中，对象传入传出都各有一次对象的拷贝，浪费了不必要的时间和空间。

【解决方案】传递对象时更改为指针传递，避免不必要的开销。

```
Matrix multiply(const Matrix* matrix2);
```

【难度系数】 

【难点分析】指针和引用的过程中容易出错，从而导致程序异常退出(exit 3)。

【优化效果】

测试用例：两个1024*1024阶密集方阵

矩阵乘法计算时间：12000ms-13000ms

详情请见以下的截图：

```
D:\CLionProjects\cppPro1\cmake-build-debug\cppPro1.exe
Data IO Time:63ms
Start!
12545ms

Process finished with exit code 0
```

【参考链接】https://www.cnblogs.com/whale90830/p/10536281.html

2.简单函数inline处理（Requirement4）

【问题分析】C++在函数调用的时候会有额外的开销。而这时如果函数体较为短小的话，函数调用的开销就会比较大。如果将简单函数使用inline关键字处理并在头文件里加以实现的话，可以进一步缩短运行的时间。

【解决方案】在两个函数体较小的函数添加inline关键字修饰。

```
inline void initiate(unsigned int r,unsigned int c);
inline bool setSelected(int i, int j, float value);
```

【难度系数】 

【难点分析】这是于仕琪老师在Lecture7中讲授过的关于inline关键字的知识点，故无过多难点

【优化效果】

测试用例：两个1024*1024阶密集方阵

矩阵乘法计算时间：11700ms-12500ms

详情请见以下的截图：

```
D:\CLionProjects\cppPro1\cmake-build-debug\cppPro1.exe
Data IO Time:62ms
Start!
11853ms

Process finished with exit code 0
```

【参考链接】https://blog.csdn.net/u011760195/article/details/100828112

3.改进C/C++的IO效率（Requirement4）

【问题分析】
①C++中为了向下兼容，保持了和C语言标准输入输出（如scanf，printf等等）的输入输出流的绑定。cin和cout存在输入输出流的缓冲区，在使用时需要先将内容放入缓冲区，然后再进行IO操作。
②cout的endl输出换行操作会刷新清空输出缓冲区，又造成了多余的时间开销。
③参考python的numpy矩阵库，可以在默认情形下对于过大的矩阵默认省略输出部分元素，从而节约IO时间
【解决方案】
①关闭cin、cout、cerr的同步绑定关系
②（废弃）关闭与stdio的默认IO同步关系（废弃原因：取消绑定后，导致命令行输出顺序异常，对用户体验感极为不利，故废弃）
③设置矩阵类的默认省略显示模式

【代码实现】
main.cpp中的
```
inline void improveCIO();
```

CMatrix.cpp中的
```
void printCM() const;
```
【难度系数】 
【难点分析】这是数据结构与算法题目中常见的优化策略，无太大难度。
【参考链接】http://www.hankcs.com/program/cpp/cin-tie-with-sync_with_stdio-acceleration-input-and-output.html

4.O3优化编译选项（Requirement4）

【问题分析】之前所做的优化，都是基于O0（默认编译选项）进行的运行，，并没有完全发挥gcc编译器的性能，导致运行时间较长。
【解决方案】开启gcc的O3编译选项（最大限度的优化），通过增加编译代码的代码量来换取运行时间的缩短。
【代码实现】增加一句代码即可调用gcc编译器的O3优化选项：
```
#pragma GCC optimize(3,"Ofast","inline")
```
【难度系数】 
【难点分析】这是于仕琪老师在Lecture8中讲授过的关于O3优化的知识点，因此无太大难度。由于O3是编译选项，会有多个文件一起编译，因此建议在CMatrix.h和CMatrix.cpp上也增加该编译选项的描述语句。
【优化效果】
测试用例：两个1024*1024阶密集方阵
矩阵乘法计算时间：11000ms-12000ms
详情请见以下的截图：

```
D:\CLionProjects\cppPro1\cmake-build-debug\cppPro1.exe
Data IO Time:38ms
Start!
11853ms

Process finished with exit code 0
```

【参考链接】https://blog.csdn.net/qq_41289920/article/details/82344586

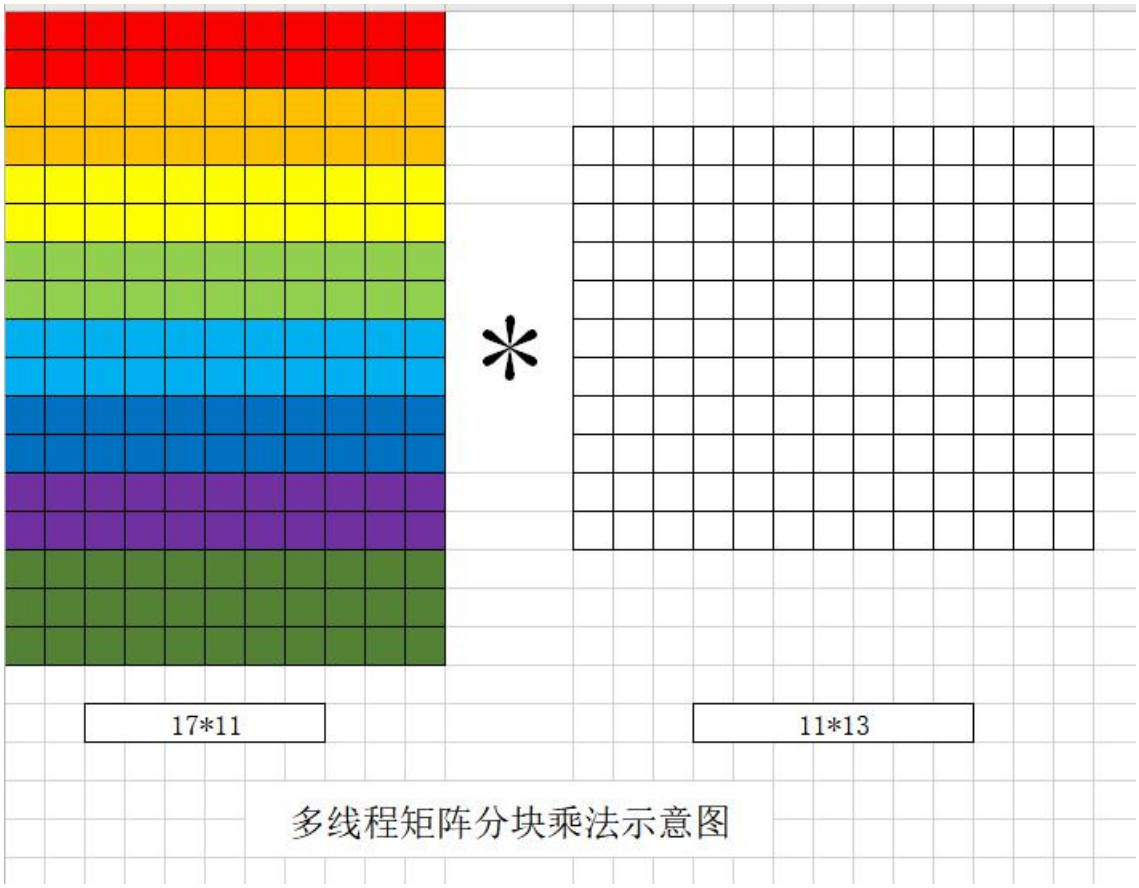
5.多线程并行计算&数组存储运算结果（Requirement4）

【问题分析】该问题的分析类似Assignment3的向量点积。在本次运算中，最坏情况时间复杂度为$O(n^3)$，但是各个部分的元素运算是互不干扰的。因此在矩阵非常巨大的时候，采用多线程并行计算可以显著提高效率

【提示】同Assignment3。`std::thread::hardware_concurrency()`函数可以得到本机的CPU核数，对于调整合适的线程数目有帮助。

【解决方案】建立thread类型的vector线程池，然后把多个长度的向量运算要求逐一分发下去，并且通过join()函数让主线程等待子线程运行完毕。利用数组存储运算结果，再将运算结果合并起来，可以有效避免线程多次同时访问修改造成的data race问题。

【线程分配方案】先将本矩阵的行分为等同于CPU核数的数量（有余数的话则让最后一个线程多承担一些任务）。然后将分块的矩阵分别计算输出矩阵的元素值即可。如下图所示。



多线程矩阵分块乘法示意图

【代码实现】CMatrix.cpp中的

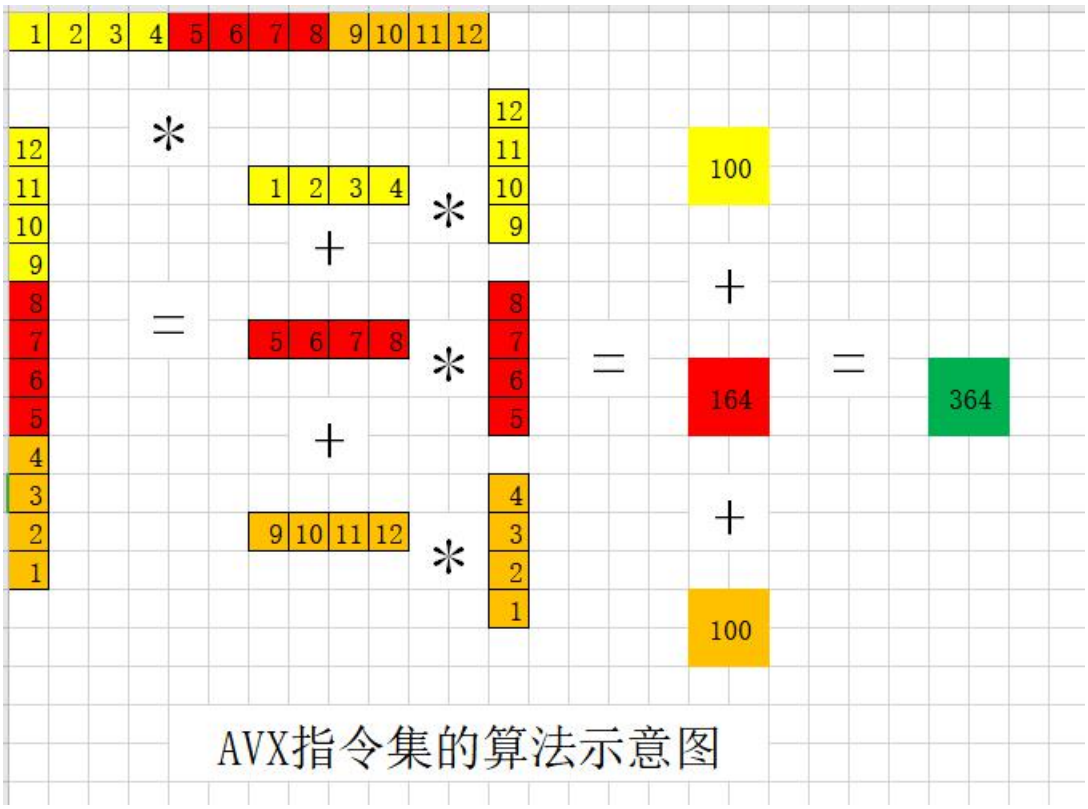
```
multipleMultiply(&cm);
static CMatrix subMultiply
(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float*
results);
```

【难度系数】 
【难点分析】多线程有多个难点需要克服：
①多线程之间数据加锁，避免data race造成数据错误；
②多线程间每个线程之间的任务量的分配；
③主线程对于其他子线程的等待操作；
④子线程的合理数目的数量确定（少了不能发挥最大效益，多了会因为互相通讯而拖慢时间）；
⑤多线程在O3编译优化选项开启后才有显著优势，否则会慢于单线程；
⑥避免多线程之间加锁的时候错误，导致线程死锁；
⑦分配下去的子任务的正确执行与实现；
⑧数组存储结果时需要有正确的合并过程；
【优化效果】
测试用例：两个1024*1024阶密集方阵
矩阵乘法计算时间：150ms-200ms
详情请见以下的截图：

```
A*B:
9.76331e+31 9.60515e+31 9.81774e+31 ... 9.3981e+31 9.70692e+31 9.58284e+31
9.62038e+31 9.47557e+31 9.72625e+31 ... 9.35326e+31 9.5947e+31 9.55593e+31
9.95429e+31 9.87513e+31 1.02818e+32 ... 9.91392e+31 1.01676e+32 9.94593e+31
...
9.78514e+31 9.86587e+31 1.01904e+32 ... 9.87841e+31 1.00212e+32 9.9791e+31
9.67784e+31 9.77025e+31 9.99168e+31 ... 9.80608e+31 9.92318e+31 9.81524e+31
9.86669e+31 9.69855e+31 1.00785e+32 ... 9.89166e+31 1.01545e+32 9.80941e+31
Multiplication Time:176ms
Do you want to calculate again? Print[Y] to continue.
```

6.使用指令集加速计算（Requirement4）

【问题描述】在加法或者乘法运算时，我们每次都是各访问1个元素，然后进行一次乘法，或许可能有更好的策略来实现这一过程。
【解决方案】Inter提供的指令集允许我们对多个元素同时进行乘法操作，从而节约时间。利用Inter提供的指令集可以更高效地实现这一功能。（可以考虑不严格对位对齐的指令集代码来提高写代码时的容错率）（AVX指令集算法如下图所示，相同颜色的方块表示组成的一个指令集向量，将会同时进行乘法运算）



【代码实现】CMatrix.cpp中的

```
static CMatrix subMultiply
(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float*
results);
```

【难度系数】 

【难点分析】边界条件的考虑，加载矩阵元素时的各个数字不能越界；AVX指令集的熟悉与使用过程；计算完成后将结果存储返回的流程；

【优化效果】

测试用例：两个1024*1024阶密集方阵

矩阵乘法计算时间：90ms-110ms

详情请见以下的截图：

```
A*B:
7.64435e+31 7.82947e+31 7.96068e+31 ... 7.32637e+31 7.96357e+31 8.22159e+31
7.60179e+31 7.72473e+31 7.78484e+31 ... 7.25971e+31 8.02819e+31 8.08008e+31
7.83584e+31 8.05161e+31 7.91889e+31 ... 7.55854e+31 7.99956e+31 8.2955e+31
...
8.26326e+31 8.31425e+31 8.41162e+31 ... 8.08945e+31 8.49987e+31 8.63805e+31
7.77395e+31 8.04318e+31 8.07401e+31 ... 7.67624e+31 8.32265e+31 8.4974e+31
7.98551e+31 7.98479e+31 8.01805e+31 ... 7.66264e+31 8.29007e+31 8.22911e+31
Multiplication Time:94ms
```

7.提升随机数的生成速度（Requirement4）

（该部分同Assignment3的随机数生成方法）

【问题分析】本次报告的大数据测试用例都是通过现场生成的随机数。随机数生成可以调用rand()函数来生成伪随机数，调用srand()函数来设置随机数的种子。但是rand()函数的运行效率有待提高，如在第一版调用rand函数时需要60~80ms来生成100万个随机数。

【解决方案】采用Xorshift算法，先只生成2个系统时间设置种子的伪随机数，后面通过一些算数运算来生成随机数，从而提高生成随机数的效率。

【代码实现】以下方法可以实现：

```
inline float randomGenerate();
```

【难度系数】 

【难点分析】Assignment3已经有过相应的实现，故实现起来是轻车熟路。这个过程中要自己设计一个比较随机的运算方法，以及要增加全局变量来保证下一次的伪随机生成。

【优化效果】

测试用例：两个1024*1024阶矩阵相乘

随机数生成时间：10ms~15ms

详情请见以下的截图：

```
Please input for m
1024
Please input for n
1024\
Please input for k
1024
If you want to make matrix1 be a random matrix? If so, please print[Y].
Y
If you want to make matrix2 be a random matrix? If so, please print[Y].
Y
415491503 481362777 241742156
A:
B:
Data IO Time:14ms
```

【参考链接】https://blog.csdn.net/u012440684/article/details/50513423

8.空间使用与回收（Requirement4）

【问题分析】之前提出过循环计算功能。而在初始化时，都需要利用new关键字开辟一个新的内存空间。如果先前的内存空间不进行释放，则容易导致内存泄漏，程序异常退出等等。

【解决方案】在下面函数中，每次都有delete对应矩阵的指针即可。

【代码实现】请见

```
inline void initiate();
```

【难度系数】 

【难点分析】这是于仕琪老师Lecture5上课所讲的东西。因此直接应用难度不是很大。

【优化效果】

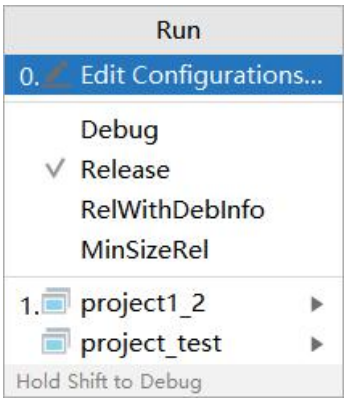
无时间上的明显优化，但是实现了在一次运算完后对内存进行释放，提高了系统空间的利用率，有效避免了内存泄漏问题，保证了程序能够正常地循环利用。

9.编译模式由Debug模式转为Release模式（Requirement4）

【问题分析】在IDE的debug选项状态下，c++的程序运行速度很慢，没有发挥出潜在的潜能。切换编译模式可以进一步提高效率。

【解决方案】切换IDE的编译选项由Debug转为Release。

【具体操作】点击CLion中上方的run选项，然后点击Run（第4个），然后在弹出的菜单栏里勾选release即可。



【难度系数】 
【难点分析】需要知道Debug模式和Release模式的区别，以及在IDE中如何切换这两种模式。
【优化效果】
测试用例：两个1024*1024密集型随机矩阵
计算时间：80ms~100ms

```
A*B:
8.09556e+31 7.77944e+31 8.02367e+31 ... 8.21949e+31 8.00502e+31 7.98644e+31
7.86804e+31 7.95202e+31 7.99469e+31 ... 8.19454e+31 7.9799e+31 7.79146e+31
8.11428e+31 8.14519e+31 8.21406e+31 ... 8.05191e+31 7.98172e+31 7.99821e+31
...
7.92415e+31 8.02351e+31 7.952e+31 ... 7.90258e+31 7.85356e+31 7.81962e+31
8.30545e+31 8.01743e+31 8.00264e+31 ... 8.20387e+31 8.09884e+31 7.91998e+31
7.95858e+31 7.81111e+31 7.93235e+31 ... 7.88134e+31 7.76807e+31 7.95144e+31
Multiplication Time:89ms
```

【参考链接】https://blog.csdn.net/m0_38068229/article/details/89356939

10.更换矩阵乘法的循环次序（Requirement4）

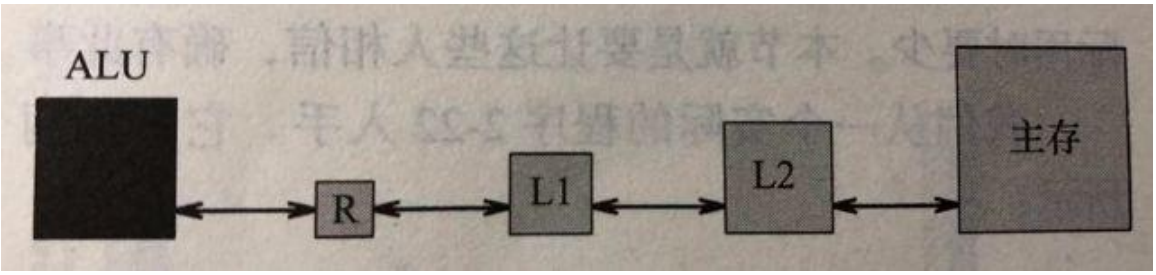
【问题背景】在写矩阵乘法的实现时，我们很自然地可以写出类似这样的代码：

```
for (int i = 0; i < m; i++){//for-loop-1
    for (int j = 0; j < k; j++){//for-loop-2
        for (int s = 0; s < n; s++){//for-loop-3
            c[i][j] = c[i][j] + a[i][s]*b[s][j];
        }
    }
}
```

但是从循环的角度来看，这个代码效率并不是最高的
【问题分析】事实上，我们如果交换矩阵循环的次序，并使用中间变量暂时存储的话，效率可以继续提高：

```
for (int i = 0; i < m; ++i){//for-loop-1
      for (int s = 0; s < n; ++s){//for-loop-3
          double value=c[i][j];
          for (int j = 0; j < k; ++j){//for-loop-2
              value += a[i][s]*b[s][j];
           }
           c[i][j]=(float)value;
      }
```

}
原因分析：这是源于计算机系统的缓存命中问题。计算机的系统结构的一部分示意图如下所示：
L1：一级缓存
L2：二级缓存
R：寄存器
ALU：算术逻辑单元



交换了循环次序2并将c[i][j]存储到局部变量以后，我们对于a矩阵和b矩阵的缓存命中率可以大大增加，从而减少ab矩阵缓存在循环过程中丢失的可能，这样一来，就可以提高循环计算效率。
【难度系数】☆☆☆
【难点分析】需要考虑循环次序对于矩阵乘法性能的影响；需要考虑计算机底层系统的缓存命中问题
【函数实现】CMatrix.cpp中的

```
static CMatrix subMultiply(const CMatrix* cm1,const CMatrix*
cm2,int start,int end,float* results);

CMatrix singleMultiply(const CMatrix* cm) const;
```

【优化效果】
测试用例：两个1024*1024密集型随机矩阵
计算时间：75ms~100ms
具体请见以下截图：

```
A*B:
8.32295e+31 8.07411e+31 8.18923e+31 ... 8.13736e+31 7.91552e+31 8.23775e+31
8.37785e+31 8.10113e+31 8.48765e+31 ... 8.4252e+31 8.31637e+31 8.31605e+31
7.96494e+31 7.9884e+31 8.03931e+31 ... 8.01633e+31 7.87726e+31 8.3322e+31
...
8.24927e+31 8.02246e+31 8.08662e+31 ... 8.27701e+31 8.05566e+31 8.47229e+31
8.28567e+31 8.1616e+31 8.3193e+31 ... 8.48687e+31 8.23167e+31 8.64334e+31
8.34132e+31 8.00052e+31 8.03056e+31 ... 8.24443e+31 7.97001e+31 8.37005e+31
Multiplication Time:82ms
Do you want to calculate again? Print[Y] to continue.
```

【参考链接】http://codingdict.com/questions/95178

11.根据矩阵的特殊性决定运算策略（Requirement4）

【时间复杂度分析】
设输入矩阵A(m*n)，B(n*k)，输出矩阵C(m*k)
①密集情形：**（最坏情形）**
在计算矩阵A与矩阵B鉴于每次计算输出矩阵C的C[i][j]元素的时候需要遍历所有的A[i][k]和B[k][j]，然后C共m*k个元素，故计算C=A*B的时间复杂度为$O(mnk)$。

②稀疏情形：
在某一个矩阵零元素较多的时候，可以考虑遇到零元素的时候直接continue而不继续循环。总体上时间复杂度仍为$O(n^3)$，但是实际运算时间比密集型大大减少。
③对角阵情形：在某一个矩阵为对角阵的情形下，只需访问对角线上的元素，然后分别乘以另一矩阵的行/列，因此时间复杂度为$O(n^2)$。
Tips：考虑双对角阵相乘的情形，此时只需要对角线上元素分别相乘，时间复杂度为$O(n)$。
④向量情形：
该情形的前提为m=1&&k=1，则该种情形回到了Assignment3的向量点积模式，此时复杂度为$O(n)$
⑤零矩阵情形：**（最优情形1）**
该情形的前提为某一个矩阵所有元素均为0，则该种情形只需建立一个默认矩阵即可，运算时间复杂度为$O(1)$。
⑥单位阵情形：**（最优情形2）**
该情形的前提为某一个矩阵为单位阵，则该种情形只需将另一个矩阵输出即可，运算时间复杂度为$O(1)$。

优化情形一：稀疏阵

【稀疏矩阵定义】在矩阵中，若数值为0的元素数目远远多于非0元素的数目，并且非0元素分布没有规律时，则称该矩阵为稀疏矩阵。
【优化策略】
①更换循环次序为上述中优化过的for-loop次序
②增加条件判断，若a[i][s]==0则直接continue，减少循环嵌套的次数
（因为后面乘起来都是0，没有意义）
【代码实现】

```
static CMatrix subMultiply
(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float*
results);
```

【优化效果】
测试用例：两个1024*1024稀疏矩阵
计算时间：15~20ms
Test case #8:
请在test.cpp中调用此函数以测试

```
void test_case8();
```

$$\begin{bmatrix} 4 & 2 & 0 & \dots & 0 & 0 & 0 \\ 2 & 4 & 0 & \dots & 0 & 0 & 0 \\ \dots & & & \dots & & & \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 3 & 0 & \dots & 0 & 0 & 0 \\ 3 & 1 & 0 & \dots & 0 & 0 & 0 \\ \dots & & & \dots & & & \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 10 & 14 & 0 & \dots & 0 & 0 & 0 \\ 14 & 10 & 0 & \dots & 0 & 0 & 0 \\ \dots & & & \dots & & & \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

Screen-short for case #5:

```
A:
4 2 0 ... 0 0 0
2 4 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
B:
1 3 0 ... 0 0 0
3 1 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
A*B:
10 14 0 ... 0 0 0
14 10 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
20ms

Process finished with exit code 0
```

优化情形二：对角阵

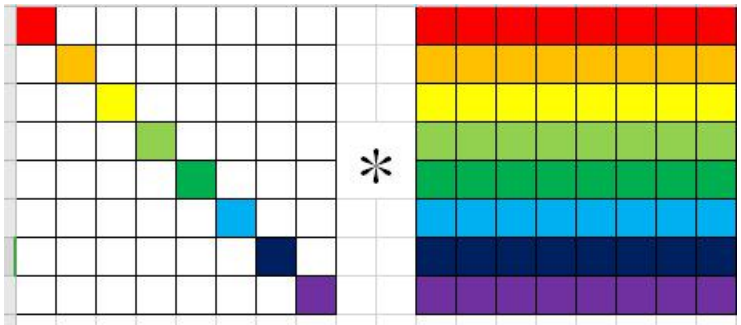
【对角矩阵定义】对角矩阵(diagonal matrix)是一个主对角线之外的元素皆为0的矩阵。

【优化策略】

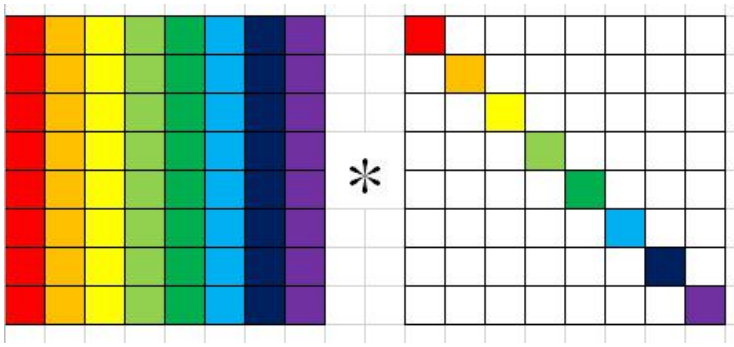
①利用int和bool变量实现对于对角元和非对角元的非零元数量的动态监测：

```
private:
    bool isDiagonal= false;
    int nonZeroNumberNotMain=0;
    int nonZeroNumberMain=0;
```

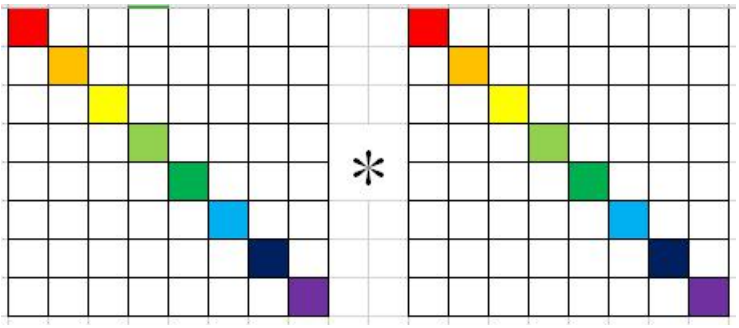
②在某一个矩阵为对角阵时，算法可以精简优化：此时对于对角矩阵只需访问对角元上的特定元素即可，主要有以下三种，如下图所示：（相同颜色的方格表示需要访问并且相乘的元素）



（第一个矩阵为对角阵）



（第二个矩阵为对角阵）



（两个矩阵都为对角阵）

【代码实现】CMatrix.cpp中的

```
static CMatrix diagonalMatrixMultiplication
(const CMatrix* cm1,const CMatrix* cm2);
```

【优化效果】

Test case #9:

测试用例：两个1024*1024矩阵，其中一个为对角阵，另一个为稠密阵

计算时间：28~30ms

请在test.cpp中调用此函数以测试

```
void test_case9();
```

$$\begin{bmatrix} 3 & 3 & \dots & 3 & 3 \\ 3 & 3 & \dots & 3 & 3 \\ \dots & & & & \\ 3 & 3 & \dots & 3 & 3 \\ 3 & 3 & \dots & 3 & 3 \end{bmatrix} * \begin{bmatrix} 2 & 0 & \dots & 0 & 0 \\ 0 & 2 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 2 & 0 \\ 0 & 0 & \dots & 0 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 6 & \dots & 6 & 6 \\ 6 & 6 & \dots & 6 & 6 \\ \dots & & & & \\ 6 & 6 & \dots & 6 & 6 \\ 6 & 6 & \dots & 6 & 6 \end{bmatrix}$$

Screen-short for case #9:

```
A:
3 3 3 ... 3 3 3
3 3 3 ... 3 3 3
3 3 3 ... 3 3 3
...
3 3 3 ... 3 3 3
3 3 3 ... 3 3 3
3 3 3 ... 3 3 3
B:
2 0 0 ... 0 0 0
0 2 0 ... 0 0 0
0 0 2 ... 0 0 0
...
0 0 0 ... 2 0 0
0 0 0 ... 0 2 0
0 0 0 ... 0 0 2
A*B:
6 6 6 ... 6 6 6
6 6 6 ... 6 6 6
6 6 6 ... 6 6 6
...
6 6 6 ... 6 6 6
6 6 6 ... 6 6 6
6 6 6 ... 6 6 6
30ms

Process finished with exit code 0
```

Test case #10:
请在test.cpp中调用此函数以测试

```
void test_case10();
```

测试用例：两个1024*1024矩阵，两个均为对角阵
计算时间：8~10ms

$$\begin{bmatrix} 2 & 0 & \dots & 0 & 0 \\ 0 & 2 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 2 & 0 \\ 0 & 0 & \dots & 0 & 2 \end{bmatrix} * \begin{bmatrix} 3 & 0 & \dots & 0 & 0 \\ 0 & 3 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 3 & 0 \\ 0 & 0 & \dots & 0 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 0 & \dots & 0 & 0 \\ 0 & 6 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 6 & 0 \\ 0 & 0 & \dots & 0 & 6 \end{bmatrix}$$

Screen-short for case #10:

```
D:\CLionProjects\project1_2\cmake-bu
A:
3 0 0 ... 0 0 0
0 3 0 ... 0 0 0
0 0 3 ... 0 0 0
...
0 0 0 ... 3 0 0
0 0 0 ... 0 3 0
0 0 0 ... 0 0 3
B:
2 0 0 ... 0 0 0
0 2 0 ... 0 0 0
0 0 2 ... 0 0 0
...
0 0 0 ... 2 0 0
0 0 0 ... 0 2 0
0 0 0 ... 0 0 2
A*B:
6 0 0 ... 0 0 0
0 6 0 ... 0 0 0
0 0 6 ... 0 0 0
...
0 0 0 ... 6 0 0
0 0 0 ... 0 6 0
0 0 0 ... 0 0 6
9ms

Process finished with exit code 0
```

优化情形三：向量情形

（该情形为m==1&&k==1的特殊情形，在Assignment3已经给出了实现，故不在此赘述）

【代码实现】CMatrix.cpp中的

```
static CMatrix vectorMultiply
(const CMatrix* vector1,const CMatrix* vector2);

static void subVectorMultiply
(const CMatrix* cm1,const CMatrix* cm2,int start,int end,float*
results,int i);
```

优化情形四：零矩阵/单位阵情形

【零矩阵定义】零矩阵，零矩阵即所有元素皆为0的矩阵。

【单位阵定义】主对角线上的元素都为1，其余元素全为0的方阵称为单位矩阵。

【优化策略】

①利用int和bool变量实现对于对角元和非对角元的非零元和非一元的数量的动态监测：

```
private:
    bool isSquare= false;
    bool isDiagonal= false;
    bool isIdentical=false;
    bool isZeroMatrix= false;
```

```
    int nonZeroNumberNotMain=0;
    int nonZeroNumberMain=0;
    int nonOneNumberMain=0;
    int nonZeroNumber=0;
```

②在某一个矩阵为单位阵或者零矩阵时，算法可以大大缩短时间（因为无需计算了，只需生成符合要求的矩阵即可）。

【代码实现】

```
static CMatrix identicalMatrixMultiplication
(const CMatrix* cm1,const CMatrix* cm2);//单位矩阵乘法

static inline CMatrix zeroMatrixMultiplication
(const CMatrix* cm1,const CMatrix* cm2);//零矩阵乘法
```

Test case #11:

请在test.cpp中调用此函数以测试

```
void test_case11();
```

测试用例：两个1024*1024矩阵，一个为单位阵，另一个为密集阵

计算时间：8~11ms

$$\begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} * \begin{bmatrix} 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \\ \dots & & & & \\ 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \end{bmatrix} = \begin{bmatrix} 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \\ \dots & & & & \\ 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \end{bmatrix}$$

Screen-short for case #11:

```
A:
1 0 0 ... 0 0 0
0 1 0 ... 0 0 0
0 0 1 ... 0 0 0
...
0 0 0 ... 1 0 0
0 0 0 ... 0 1 0
0 0 0 ... 0 0 1
B:
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
...
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
A*B:
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
...
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
8ms

Process finished with exit code 0
```

Test case #12:

请在test.cpp中调用此函数以测试

```
void test_case12();
```

测试用例：两个1024*1024矩阵，一个为零矩阵，另一个为密集阵

计算时间：8~11ms

$$\begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} * \begin{bmatrix} 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \\ \dots & & & & \\ 996 & 996 & \dots & 996 & 996 \\ 996 & 996 & \dots & 996 & 996 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

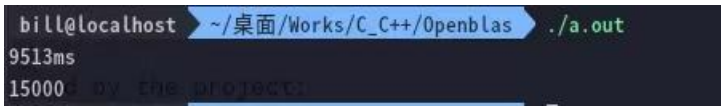
Screen-short for case #12:

```
A:
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
B:
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
...
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
996 996 996 ... 996 996 996
A*B:
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
9ms

Process finished with exit code 0
```

【200M大矩阵的运算计时】

【总体描述】
密集情形下效率不如OpenBLAS库，但是特殊情形下速度可以比OpenBLAS库要快很多。
【OpenBLAS】测评环境：12核Linux系统



（感谢提供测试平台的的11913008谢岳臻同学）
两个15000阶矩阵相乘用时：9500ms左右

1.密集阵*密集阵

Test case #13:
请在test.cpp中调用此函数以测试

```
void test_case13();
```

测试用例：两个15000阶随机数密集大矩阵相乘
测试用时：391738ms=391s（时间长于openblas）

Screen-short for case #13:

```
A:
41 6334 19169 ... 16359 4606 25357
8970 14252 31360 ... 2405 4903 8902
3154 8543 9519 ... 16964 7373 15888
...
1609 29814 18913 ... 7936 21148 13082
3635 2139 15063 ... 5923 12331 12311
2428 21429 4356 ... 13621 6464 31716
B:
18467 26500 15724 ... 20141 15052 20860
8320 20912 6954 ... 20670 14406 14555
19830 16859 8328 ... 29951 3862 5844
...
9771 7664 23710 ... 23149 12382 18956
13178 11716 8039 ... 10016 9138 23856
10878 31115 10075 ... 28247 17221 18737
A*B:
4.05088e+12 4.05257e+12 4.06331e+12 ... 4.02889e+12 4.12225e+12 4.01761e+12
4.03049e+12 4.02575e+12 4.0551e+12 ... 4.00574e+12 4.09151e+12 3.99482e+12
4.06043e+12 4.04549e+12 4.05553e+12 ... 4.03042e+12 4.08329e+12 4.0281e+12
...
4.01966e+12 4.02249e+12 4.05036e+12 ... 4.01324e+12 4.07949e+12 3.98626e+12
4.033e+12 4.05412e+12 4.06596e+12 ... 4.01877e+12 4.08621e+12 4.00963e+12
3.98594e+12 3.9887e+12 4.03962e+12 ... 3.95338e+12 4.04801e+12 3.976e+12
391738ms

Process finished with exit code 0
```

2.密集阵*稀疏阵

Test case #14:
请在test.cpp中调用此函数以测试
```
void test_case14();
```
测试用例：两个15000阶大矩阵相乘，一个为密集随机矩阵，另一个为稀疏矩阵
测试用时：6441ms（时间短于openblas）

Screen-short for case #14:

```
A:
0 1 2 ... 0 0 0
3 4 5 ... 0 0 0
6 7 8 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
B:
41 18467 6334 ... 6116 20934 25546
32562 18255 13291 ... 15052 25357 20860
8970 8320 14252 ... 7104 16659 21573
...
15266 3331 24517 ... 5539 20757 8636
8514 7915 32560 ... 27136 28593 21074
19395 17958 711 ... 20764 8818 29005
A*B:
50502 34895 41795 ... 29260 58675 64006
175221 170021 143426 ... 114076 247525 267943
299940 305147 245057 ... 198892 436375 471880
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
6441ms

Process finished with exit code 0
```

3.密集阵*对角阵

Test case #15:
请在test.cpp中调用此函数以测试
```
void test_case15();
```
测试用例：两个15000阶大矩阵相乘，一个为密集随机矩阵，另一个为对角矩阵
测试用时：1620ms（时间短于openblas）

Screen-short for case #15:

```
A:
1 0 0 ... 0 0 0
0 2 0 ... 0 0 0
0 0 3 ... 0 0 0
...
0 0 0 ... 14998 0 0
0 0 0 ... 0 14999 0
0 0 0 ... 0 0 15000
B:
41 18467 6334 ... 6116 20934 25546
32562 18255 13291 ... 15052 25357 20860
8970 8320 14252 ... 7104 16659 21573
...
15266 3331 24517 ... 5539 20757 8636
8514 7915 32560 ... 27136 28593 21074
19395 17958 711 ... 20764 8818 29005
A*B:
41 18467 6334 ... 6116 20934 25546
65124 36510 26582 ... 30104 50714 41720
26910 24960 42756 ... 21312 49977 64719
...
2.28959e+08 4.99583e+07 3.67706e+08 ... 8.30739e+07 3.11313e+08 1.29523e+08
1.27701e+08 1.18717e+08 4.88367e+08 ... 4.07013e+08 4.28866e+08 3.16089e+08
2.90925e+08 2.6937e+08 1.0665e+07 ... 3.1146e+08 1.3227e+08 4.35075e+08
1620ms

Process finished with exit code 0
```

4.向量*向量

Test case #16:
请在test.cpp中调用此函数以测试

```
void test_case16();
```

测试用例：两个向量相乘，一个为1*200M，另一个为200M*1
测试用时：98ms（时间短于openblas）

Screen-short for case #16:

```
A:
1 2 3 ... 2e+08 2e+08 2e+08
B:
2e+08
2e+08
2e+08
...
3
2
1
A*B:
1.33333e+24
98ms

Process finished with exit code 0
```

5.密集阵*单位阵：

Test case #17:
请在test.cpp中调用此函数以测试

```
void test_case17();
```

测试用例：两个15000阶大矩阵相乘，一个密集型随机矩阵，另一个为单位矩阵
测试用时：10ms（时间短于openblas）

Screen-short for case #17:

```
D:\CLionProjects\project1_2\cmake-build-release\project_test.exe
A:
1 0 0 ... 0 0 0
0 1 0 ... 0 0 0
0 0 1 ... 0 0 0
...
0 0 0 ... 1 0 0
0 0 0 ... 0 1 0
0 0 0 ... 0 0 1
B:
41 18467 6334 ... 6116 20934 25546
32562 18255 13291 ... 15052 25357 20860
8970 8320 14252 ... 7104 16659 21573
...
15266 3331 24517 ... 5539 20757 8636
8514 7915 32560 ... 27136 28593 21074
19395 17958 711 ... 20764 8818 29005
A*B:
41 18467 6334 ... 6116 20934 25546
32562 18255 13291 ... 15052 25357 20860
8970 8320 14252 ... 7104 16659 21573
...
15266 3331 24517 ... 5539 20757 8636
8514 7915 32560 ... 27136 28593 21074
19395 17958 711 ... 20764 8818 29005
10ms

Process finished with exit code 0
```

6.密集阵*零矩阵

Test case #18:
请在test.cpp中调用此函数以测试

```
void test_case18();
```

测试用例：两个15000阶大矩阵相乘，一个密集型随机矩阵，另一个为零矩阵
测试用时：500ms（时间短于openblas）
（时间长于单位阵是源于开空间需要时间）

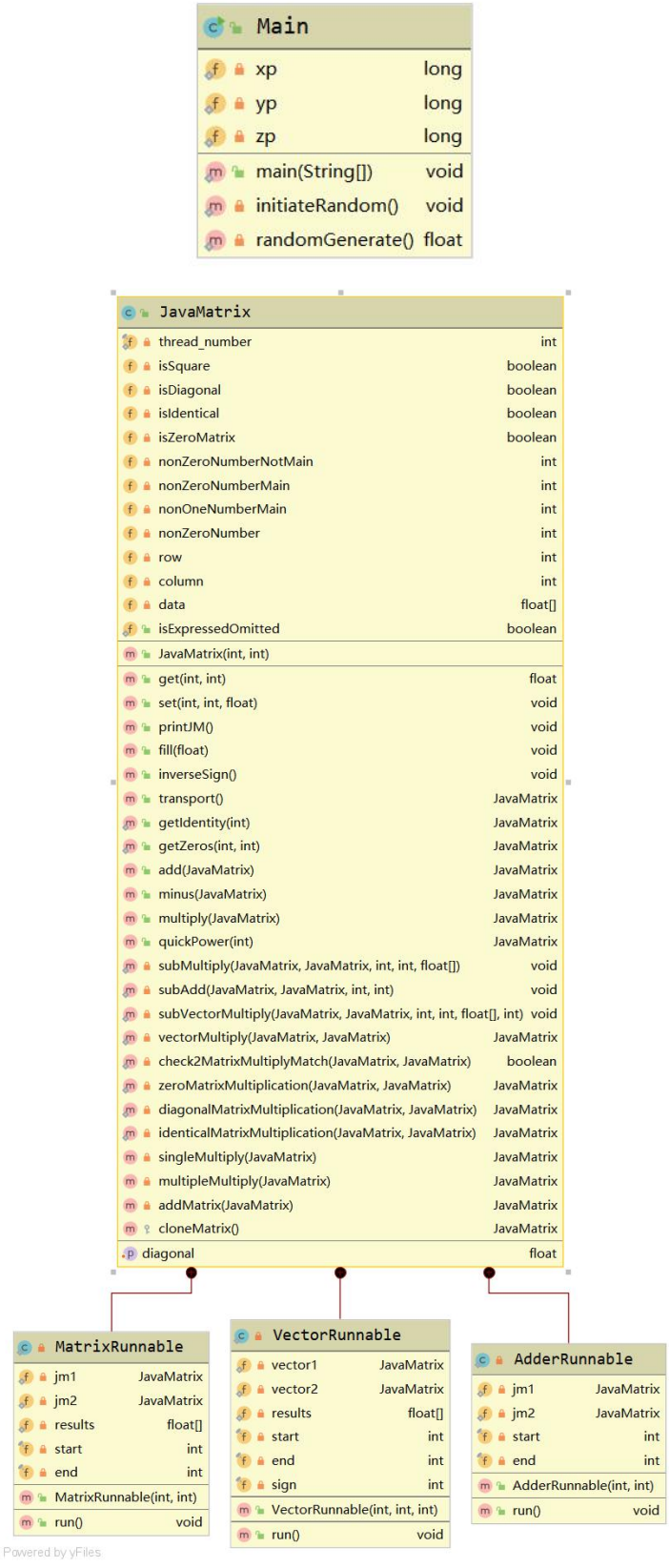
```
A:
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
B:
41 18467 6334 ... 6116 20934 25546
32562 18255 13291 ... 15052 25357 20860
8970 8320 14252 ... 7104 16659 21573
...
15266 3331 24517 ... 5539 20757 8636
8514 7915 32560 ... 27136 28593 21074
19395 17958 711 ... 20764 8818 29005
A*B:
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
...
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
0 0 0 ... 0 0 0
495ms

Process finished with exit code 0
```

【与Java语言的比较与分析】

1. 关于Java代码的简要说明

【IDE】IDEA Ultimate 2020.2
【代码整体结构】请见以下JavaMatrix.java的整体结构的截图。方法的实现与C++中的CMatrix类基本上一致。在Main.java的测试文件中，为了精简，只保留了随机数生成这一功能。



2. Java/C++两种语言的测试效果展示

为了检测的精简和高效，我们考虑以下测试用例：

【测试用例】两个5000阶随机数密集型矩阵

【Java测试用时】55911ms=55s

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
A:
-2.46846029E9 2.88869709E9 -3.19951821E9 ... -1.44513434E9 -4.0264297E9 -7.3176685E8
1.3661664E9 3.36983424E9 -3.04347443E9 ... 7.4790608E7 3.59197184E9 -8.4728429E8
1.85658227E9 -1.35944205E9 -4.15484058E9 ... 2.97628979E9 2.99658957E9 -4.17731072E9
...
2.43904102E9 -1.56662605E9 -6.6688819E8 ... 2.08450096E8 -5.34305088E8 9.9324928E7
3.01473971E9 3.3627351E9 4.04283699E9 ... -3.75249254E9 -3.7793856E9 -2.00470592E8
-4.24018125E9 2.19771341E9 3.15282278E9 ... 3.88269107E9 5.7125242E8 1.20775194E9
B:
-2.86041139E9 -4.0426064E8 4.05811686E9 ... -2.15021824E9 -3.14098842E9 5.680928E8
-3.676296E8 1.31198374E9 2.50861798E9 ... -2.70250342E9 -5.32186368E8 2.26693427E9
6.5920256E7 2.20495539E9 3.80248422E9 ... -1.70154726E9 1.68424678E9 1.88125104E8
...
3.84950048E8 -1.70122778E9 -1.68146317E9 ... 2.44440448E9 -1.04144397E9 3.84848461E9
7.0389741E8 -1.85894451E9 4.29179443E9 ... -2.73748762E9 6.3195706E8 -3.27514592E8
3.0931136E8 -1.18893747E9 3.19856128E9 ... 2.76520755E9 3.7933289E9 -2.35895136E8
A*B:
-1.5009942E20 -9.367248E20 -2.0748615E20 ... 3.7927455E20 -3.4600706E20 1.994483E20
-3.8098104E19 8.975976E19 2.5610138E20 ... 2.5613213E20 -2.4226184E20 -3.3775706E20
-4.8267857E18 4.5941273E19 1.61284E20 ... -2.2261077E20 -3.9741174E20 6.3779886E20
...
1.0175972E20 -4.9104893E20 -2.9758284E20 ... 9.810117E20 7.75575E20 2.3355075E20
-1.2998471E20 5.026516E20 1.540834E20 ... 1.1840555E20 -6.0256896E20 1.4692679E21
4.5521414E20 1.8286458E20 -6.843616E20 ... -1.7272455E20 9.297978E20 -1.767634E20
55911ms
```

【C++测试用时】12604ms=12s

project_test

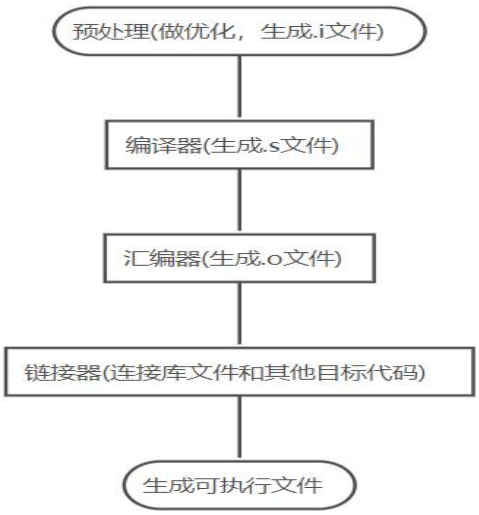
```
19
A:
41 6334 19169 ... 31126 23296 31880
19484 17194 17059 ... 12953 20641 23533
3388 3147 20455 ... 16359 4606 25357
...
6342 15018 30714 ... 28638 9248 26703
4330 29554 30364 ... 21795 20925 13865
24512 16541 30166 ... 20865 27722 24218
B:
18467 26500 15724 ... 27557 1546 18796
22693 10605 6838 ... 29360 8669 6051
16112 28420 5574 ... 20141 15052 20860
...
3346 22686 9704 ... 13365 21305 9502
25487 3714 1569 ... 3975 26794 22025
16199 23309 22240 ... 27492 11160 3581
A*B:
1.34342e+12 1.35055e+12 1.35086e+12 ... 1.3524e+12 1.32084e+12 1.3491e+12
1.36421e+12 1.35797e+12 1.37541e+12 ... 1.37207e+12 1.36192e+12 1.36746e+12
1.34092e+12 1.34641e+12 1.34852e+12 ... 1.35014e+12 1.33148e+12 1.34157e+12
...
1.32905e+12 1.33523e+12 1.35218e+12 ... 1.34362e+12 1.3248e+12 1.34397e+12
1.33167e+12 1.3343e+12 1.33495e+12 ... 1.33362e+12 1.30944e+12 1.32332e+12
1.34283e+12 1.35578e+12 1.36446e+12 ... 1.36445e+12 1.34131e+12 1.33769e+12
12604ms

Process finished with exit code 0
```

3. 具体的原因与分析

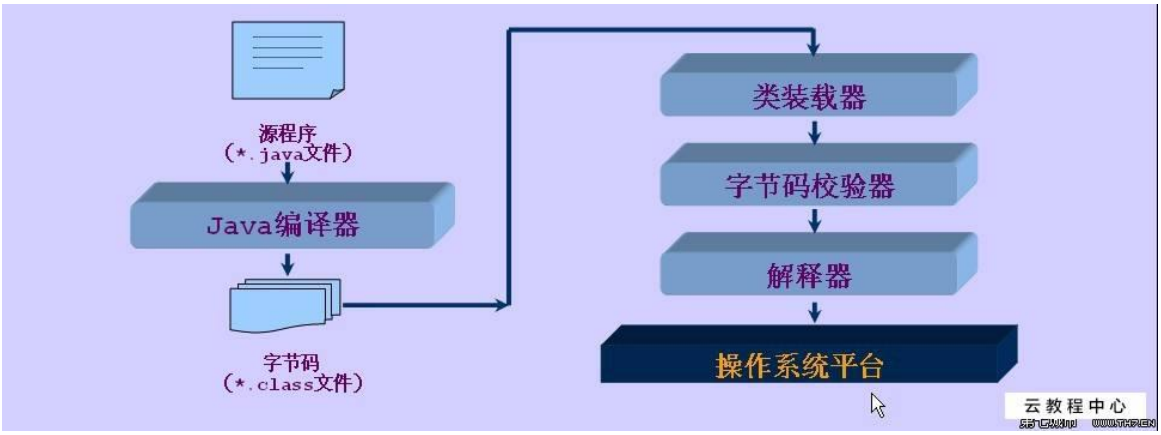
1. Java和C++两种语言的编译过程有所不同

C++的编译过程如左图所示：



在这个过程中，G++编译器可以将我们的源代码编译为机器可执行的二进制码，从而直接执行，效率较高；

而Java的编译过程如下图所示。对于 Java 语言，java 编译器会先将 java 源码解析成字节码（独立于 CPU 架构，CPU不能识别该码），然后字节码会被 JVM 加载然后再编译成当前 cpu 架构所对应的二进制代码（机器指令的集合）然后再执行。



这样一来，Java就比C++多了一层字节码处理以实现机器平台的较好兼容性，但是因此付出的代价就是比C++直接编译为机器码要慢了一些。

2. Java和C++两种语言对于对象的创建机制有所不同

Java的对象创建是通过new关键字实现的，具体对象保存在了堆里面，而栈就只有一个对象的引用，这样调用对象的方法，获取对象的内容时就需要从栈到堆的访问，有额外的开销。
而C++的很多变量直接保存在栈里面，直接访问可以提升效率。

3. Java和C++在运行过程中的检查机制有所不同

Java的虚拟机JVM为了程序的稳定性，在运行时有很多的检查，如数组越界，对象回收等等，这些检查有额外的时间开销；
但是C++就有所不同，遇到内存没有过多的检查而是直接执行，这样可以有效提升语言的效率，但是代价就是程序的稳定性和安全性会降低。

【参考链接】https://blog.csdn.net/qq_33904512/article/details/105539543

【提升用户体验】

以下这些功能参考了Assignment2计算器的实现，实现思路大同小异。
主要支持以下功能：

1. 支持对用户显示欢迎信息
2. 支持对用户显示帮助信息
3. 支持用户选择随机数测试模式
4. 支持对用户显示错误信息并提示用户重新输入
5. 支持用户自由切换中英双语
6. 支持用户自由切换矩阵显示模式
7. 支持用户自由随时退出
8. 支持用户在计算完毕后对计算器重新利用而不是直接退出

1.支持对用户显示欢迎信息

用户初次运行main.cpp或者输入welcome时，即可看到欢迎界面的显示：

```
------------------------------------
Welcome using this Matrix calculator!
This calculator is mainly for C/C++ Midterm Project: Matrix Multiplication.
This calculator is the combination of Assignment2 and Assignment3, so it is much better than that in Assignment3
University: SUSTech
Course: CS205(C/C++ Program Design)
Course Instructor: Shiqi Yu
Author: YeeTone Wang
SID: 11910104
Version: 1.23
If you need help, please print help to get some helpful information.
If you need welcome, please print welcome to get this information again.
If you need exit, please print exit to get exit information.
Print omit to omit some elements of matrix when showing, if the matrix is too large.
Print all to show all elements of matrix so that you can see the elements thoroughly.
[Contact me by email: 11910104@mail.sustech.edu.cn
如果需要显示中文，请输入Chinese.
------------------------------------
```

【设计目的】
①标明作者和设计目的
②帮助用户熟悉矩阵计算器的使用
【设计思路】
在允许用户输入的几个环节增加对于"welcome"字符串的检测，如果出现特定字符串则调用相应的函数进行输出，然后用一个continue语句继续循环即可。
【代码实现】
输入环节监测：

```
void cinSize(char sign);
inline void cinElement(int r,int c,int sign);
```

输出欢迎信息：

```
void printWelcome();
```

2.支持对用户显示帮助信息

用户输入help的时候，即可获得帮助信息：

```
----------------------------------------
This is the help for the calculator user!
Author: YeeTone Wang
Version: 1.23
1.Language change:
Any time, if you like, you can input English to change the calculator into the English verison.
On the contrary, you can change it into Chinese verison by inputting Chinese.
2.Input the size of matrix:
You are asked to input 3 integer variables:m,n,k
There are 2 matrix in total: A(m*n) and B(n*k)
Hint: m,n,k∈(0,25000]
3.Choose if random:
You can let the calculator produce the random matrix through XORShift Algorithm.
Therefore, you don't need to input the element by yourself!
4.Input your elements:
This is very simple. Just follow the guideline on the command line.
----------------------------------------
```

【设计目的】帮助用户熟悉矩阵计算器的使用

【设计思路】

同“欢迎语句”部分，只需增加对于"help"字符串的检测即可。

【代码实现】

输入环节监测：

```
void cinSize(char sign);
inline void cinElement(int r,int c,int sign);
```

输出帮助信息：

```
void printHelp();
```

3.支持用户选择随机数测试模式

当用户将m,n,k三元素输入完成后，允许用户选择是否通过XorShift算法生成随机数的测试模式：

```
If you want to make matrix1 be a random matrix? If so, please print[Y].
Y
If you want to make matrix2 be a random matrix? If so, please print[Y].
Q
```

【设计目的】方便用户测试，避免多次输入带来的繁杂过程

【设计思路】显示询问信息，并利用一个string对象存储用户的输入，然后再将该对象与字符串"Y"进行匹配。

【代码实现】main.cpp中的

```
inline void checkIsUserRandom(int sign);
```

4.支持对用户显示错误信息并提示用户重新输入

```
Please enjoy your matrix calculation!
Please input for m
23I
Please input for m
You have input something wrong!
23
Please input for n
```

【设计目的】对用户的非法行为进行提示，然后提示用户进行改正。

【设计思路】发现错误时用C++的标准错误输出流cerr进行输出即可。

【代码实现】
输入环节已有相应的实现：

```
void cinSize(char sign);
inline void cinElement(int r,int c,int sign);
```

5.支持用户自由切换中英双语

```
-------------------------------------
Please enjoy your matrix calculation!
Please input for m
Chinese
中文设置成功!
请输入矩阵大小描述元素m
2
请输入矩阵大小描述元素n
English
English is set successfully!
Please input for n
```

【设计目的】帮助英文不太好的用户能够友好地使用本计算器
【设计思路】
①先修改控制台编码：

```
SetConsoleOutputCP(65001);
```

以保障输出中文的时候不会乱码。
②在各个用户的输入环节增加对于"Chinese"和"English"字符串的检测
③然后建立全局bool变量以记录用户的语言状态
④最后在输出各种信息的时候加入对于语言状态的判断即可。
【代码实现】
各个输入环节都增加了对于特定语言字符串的检测。
各个带有cout和cerr的输出环节都有针对语言状态来输出的相应的实现。

6.支持用户自由切换矩阵显示模式

```
Please enjoy your matrix calculation!
Please input for m
omit
Some elements of the matrix is omitted when showing.
Please input for m
all
All elements of the matrix will be diaplayed when showing.
Please input for m
```

【设计目的】在矩阵过大的时候采取局部省略输出，节约不必要的IO时间。另外，也可以有利于用户对于大矩阵的可视化体验。
【设计思路】
此设计参考了python的numpy矩阵库。
①矩阵类里面增加关于输出是否省略的bool变量，并设为public以允许可以直接从外部修改。

②在输出的时候增加对于行和列数目的判断
（判断规则：如果矩阵行数多于6，则输出前三行和最后三行，其余用省略号（……）表示；如果矩阵列数多于6，则输出前三列和最后三列，其余用省略号（……）表示）
【代码实现】CMatrix类中的

```
void printCM() const;
```

7.支持用户在计算完毕后对计算器重新利用而不是直接退出

```
Multiplication Time:82ms
Do you want to calculate again? Print[Y] to continue.
Y
Please input for m
1
Please input for n
2
```

【设计目的】允许用户重新使用计算器而不直接退出，一方面有利于用户观察矩阵运算结果，另一方面用户不必重新打开，有利于用户的体验感。
【设计思路】main函数中利用初始化为"Y"的string对象记录是否继续运行。利用一个while循环来判断是否继续使用计算器。
（注意：每次循环时记得释放先前使用过的空间，避免内存泄漏）
【代码实现】main.cpp中的

```
int main();
```

【额外功能实现】

该部分非本次报告中重点，因此为精简，只会给出设计思路与代码实现。

1.实现根据阶数生成单位矩阵

【设计思路】根据传入的阶数建立方阵，然后再根据阶数对对角元上的元素赋值为 1 即可

【代码实现】`static CMatrix getIdentical(int x);`

2.实现根据大小生成零矩阵

【设计思路】由于矩阵对象初始化时内部元素就为 0 了，故直接返回一个新矩阵即可。

【代码实现】`static inline CMatrix getZeros(int r,int c);`

3.实现矩阵加/减法

【设计思路】参考矩阵加减法运算法则，只需遍历所有的元素相加即可。在这个过程中，实现起来和 Assignment3 的向量是一样的，同样可以采取多线程的优化策略。

【代码实现】

加法：

```
inline CMatrix operator+(const CMatrix& cm)const;
CMatrix addMatrix(const CMatrix* cm) const;

static void subAdd
(const CMatrix* cm1,const CMatrix* cm2,int start,int end);
```

减法：

```
inline CMatrix operator-(CMatrix& cm)const;
void inverseSign();//批量反号，从而利用加法的程序
CMatrix addMatrix(const CMatrix* cm) const;

static void subAdd
(const CMatrix* cm1,const CMatrix* cm2,int start,int end);
```

4.实现矩阵转置

【设计思路】参考矩阵转置规则，将行列元素实行交换然后赋值到新矩阵即可。

【代码实现】`CMatrix transport()const;`

5.实现矩阵的对角元素赋值为统一值的对角阵

【设计思路】类似单位矩阵，直接赋值即可。

【代码实现】`void setDiagonal(float value);`

6.实现矩阵内部元素的批量填充

【设计思路】调用`std::fill()`函数即可实现快速批量赋值到`data`数组中。其中要注意对于各个矩阵描述变量的数值和类型的维护。

【代码实现】`void CMatrix::fill(float value);`

7.实现允许根据下标获取/修改矩阵内部的值

【设计思路】由于是一维数组表示二维数组，所以要用一个乘以相应的列数来实现访问元素。

【代码实现】`float get(int r,int c) const;`

8.实现判断两矩阵是否相等

【设计思路】先比较行和列是否一致，然后再逐个访问内部元素比较。

【代码实现】运算符重载

```
inline bool operator==(const CMatrix& cm)const
```

9.实现方块矩阵的快速幂

【设计思路】思路来源：CS102A-2020Fall-Assignment2-Bonus Problem
方块矩阵的快速幂的伪代码如下所示：

```
The following is the pseudocode for `Quick Power of Matrix'
Quick_Power(a,n):
    ans <= Identity(n)
    for all binary digits d of n (from right to left):
        if d is 1 then ans <= ans * a mod m
            a = a * a mod m
    return ans
```

【代码实现】运算符重载

```
inline CMatrix operator^(int x);
```

【总结】

本次期中Project的课题为矩阵乘法，是Assignment2和Assignment3的结合与升级。本次期中Project题目要求明确，在做这次project的过程中学习了很多课外的知识点：

1.类与对象的设计与使用
2.对于对象的指针的使用
3.矩阵乘法的设计与实现
4.对于用户错误输入情形的检测与识别
5.对于计算机的可用最大内存的认识
6.对于基本数据类型的精度的深刻认识
7.对于程序计时方法的理解
8.函数值传递与引用传递的区别
9.inline关键字的深刻理解
10.C/C++的提升IO效率的方法与策略
11.C/C++的多种编译选项
12.C/C++多线程分配任务
13.C/C++的指令集的使用
14.C/C++多线程避免数据冲突
15.C/C++的内存空间使用与回收
16.CLion的IDE的编译选项
17.循环次序对于计算机缓冲命中的影响
18.根据对象类型决定不同的调优策略
19.对openblas标准库的学习与认识
20.Java多线程的使用
21.Java虚拟机的内部结构的认识
22.提升用户对于产品的体验感的策略
23.根据二进制特点的数值快速幂和矩阵快速幂算法
24.C/C++运算符重载