CS205 C/ C++ Programming - Project #4

Name: 徐临风 (Xu Linfeng)

SID: 11812407

Part 1 – Analysis

This project is to design a class for matrices, which should satisfy the following requirements:

✔ This class should contain the member variable $channel$ to realize that one element of matrix storage a vector.

✔ This class can support different kinds of data types, such as **unsigned char, short, int, float, double, etc.**

✔ Constructors and destructors should be carefully designed to avoid problems of memory management.

✔ Some general operators should be overloaded to support the normal matrix computation.

✔ In particular, the assignment operator should be overloaded to avoid hard copy when data of one matrix is assigned to another ones.

✔ Apply the idea of Region of Interest (ROI) to matrices: try to revise the desired submatrix without hard copy.

✔ Try to use the optimization method of matrix multiplication in $Project\ \#3$ in this class. For user convenience, the operator $*$ can be overloaded with optimization algorithm applied.

✔ Test this class on $X86$ and ARM platform: compare the differences, in particular SSE for $X86$ and $Neon$ for ARM.

In $Project\ \#3$, we defined a structure for matrices. Some operations, such as matrix multiplication, matrix copy, memory management, are realized by corresponding functions. Moreover, the programming language we used is C, meaning that there are many convenient tools we cannot use last time.

In $Project\ \#4$, we design a class for matrices rather than structure. Therefore, the corresponding functions for matrix operation and memory management are replaced by member functions (friend functions) and constructors and destructors, respectively. Apart from the above differences, we can overload some frequently used operators, such as $<<, >>$, which is convenient for the input and output of matrices into files or onto screen.

Some detailed descriptions of idea to realize the above requirements are stated as follows:

◉ In $cv :: Mat$ in $OpenCV$, it uses **union** containing 5 pointers of different data types pointing to matrix data, which share the same memory. This is a good idea, but I cannot make it. Apart from **union**, the **class template** can also support different kinds of data, but it needs to define more member variables and thus much more space consumption. We define a **macro** $myType$ to change the data type by hand, which is not wise but can save a lot space compared with that of **class template**.

◉ (i) We define 3 constructors and 1 copy constructor to create objects:

```
Mat();
Mat(int row, int col, int channel);
Mat(int row, int col, const myType * data, int channel);
Mat(const Mat & mat); // soft copy
```

① The constructor with no parameters is to test the assignment operator $=$, and also to replace the default constructor.

② The constructor with row number and column number specified is to create matrices whose data is provided by **.txt** files, and also to create temporary objects which can be returned in matrix computation $(+, -, *)$.

③ The constructor with data pointer specified additionally is to create matrices whose data is already known.

④ The copy constructor is designed to achieve soft copy and avoid problems of memory management.

(ii) The destructor is designed to free the memory in time. As you will see in $Part\ 2$, the destructor is key to achieve soft copy.

◉ In this matrix class, we overload the operators:

```
bool operator == (const Mat & mat_cmp); // 比较矩阵元素
Mat & operator = (const Mat & mat_copy); // 矩阵复制
Mat operator + (const Mat & mat); // 矩阵加法
Mat operator - (const Mat & mat); // 矩阵减法
Mat operator * (const Mat & mat); // 矩阵乘法
Mat operator ~ (); // 矩阵转置
Mat & submat (int a, int b, string fileName); // ROI修改特定区域的矩阵数据

friend Mat operator * (myType a, const Mat & rmat); // 矩阵左数乘
friend Mat operator * (const Mat & lmat, myType a); // 矩阵右数乘

friend ostream & operator << (ostream & os, const Mat & mat); // 输出矩阵到屏幕
friend ifstream & operator >> (ifstream & ifs, const Mat & mat); // 从文件读取矩阵
friend ofstream & operator << (ofstream & ofs, const Mat & mat); // 输出矩阵到文件
```

◉ In particular, we overload the assignment operator $=$ as follows:

```
Mat & Mat::operator = (const Mat & mat_copy) // soft copy
{
    cout << "Mat & Mat::operator = (const Mat & mat_copy)" << endl;
    if (this == & mat_copy)
```

```
        return *this;
    delete this->dataptr;
    this->row = mat_copy.row;
    this->col = mat_copy.col;
    this->size = mat_copy.size;
    this->channel = mat_copy.channel;
    if (mat_copy.dataptr)
    {
        this->dataptr = mat_copy.dataptr;
        this->dataptr->addPtrCount();
    }
    return *this;
}
```

① If the object is assigned to itself, then just return the reference of itself.

② Before the assignment operation, we should delete the space that the data point refers to to avoid memory leak. Here we do not use $delete[\,]$ because the pointer dataptr does not directly point to the matrix data. To manage the memory safely, we try to use the idea of **smart pointers: shared_ptr**: define a member variable to count the number of pointers which point to the space of the data of this matrix. However, since the parameter of the assignment operator overloading member function and the copy constructor are limited by the keyword **const**, we cannot directly add one member variable to count the number of pointers pointing to the space of the data of this matrix. Therefore, we design another class:: $Data$, which contains the pointer pointing to the matrix data and the member variable ptr_count to count the number of pointers pointing to the space of the data of this matrix. The design of the class:: $Data$ is shown as follows: (more detailed analysis will be given in $Part\ 2$)

```
class Data
{
private:
    myType * dataval;
    int ptr_count;

public:
    Data(size_t length);
    ~Data();

    myType * getDataValue();
    void addPtrCount();
    void minusPtrCount();
    int getPtrCount();
};
```

③ Copy the corresponding member variables: $row, column, size, channel.$

④ Judge whether the pointer of the object you want to assign is $NULL$. If it points to the space which contains the pointer pointing to the matrix data, then we just use $=$ to assign the address of data of two matrices. In this way, two pointers point to the same space, sharing the same data, which is actually called "**soft copy**".

⑤ Surely, the direct assignment of address is easy. The difficult part is to manage the memory. After achieving data-sharing, we call the member function $addPtrCount$ of class:: $Data$ to reveal that one more pointer is pointing to this space now.

◉ Following the above idea of **soft copy**, we can realize the **ROI** function in this class. ROI is usually used in picture processing to "cut" and "revise" the particular part of the whole picture. Similarly, in our case of matirx class, we need to find the submatrix with the given indices (start index, end index). Moreover, to avoid hard copy, we can follow the above idea of **soft copy**, we just assign the address of two pointers the same, sharing the same space. Then we can revise the data of the submatrix by pointers.

The definition of the member function **ROI** is shown as follows: (more detailed analysis will be given in the related test cases in $Part\ 3$)

```
Mat & Mat::submat (int a, int b, string fileName) // 传入的文件是想要修改到目标区域的
数据
{
    if (a > this->size || b > this->size)
    {
        cout << "invalid ROI" << endl;
        return *this;
    }
    int a_row = a / this->col + 1; // 一维数组的index a对应的行数
    int a_col = a % this->col + 1; // 一维数组的index a对应的列数
    int b_row = b / this->col + 1; // 一维数组的index b对应的行数
    int b_col = b % this->col + 1; // 一维数组的index b对应的行数
    int file_row = getRowNum(fileName); // 读入文件的行数
    int file_col = getColNum(fileName); // 读入文件的列数
    int max_row = max(a_row, b_row); // 判断index a和b谁对应的行数是起点, 谁对应的列数
是起点
    int min_row = min(a_row, b_row);
    int max_col = max(a_col, b_col);
    int min_col = min(a_col, b_col);
    int size_row = max_row - min_row + 1;
    int size_col = max_col - min_col + 1;

    if (size_row * size_col != file_row * file_col) // 判断传入文件的矩阵的大小是否和
想要修改的区域大小相同
    {
        cout << "invalid ROI" << endl;
        return *this;
    }

    ifstream fin(fileName);
    Mat roi_data = Mat(file_row,file_col,1); // 将文件中的数据存入Mat类的对象中
    fin >> roi_data;
    fin.close();
    // 通过指针修改对应区域的数据
    myType * roi_ptr = this->dataptr->getDataValue(); // 指向存储数据的首地址
    int copydata_index = 0;
    for (int i = (min_row-1)*this->col+min_col-1; i <= (max_row-1)*this-
>col+max_col-1; i++)
    {
        int ptr_row = i / this->col + 1;
        int ptr_col = i % this->col + 1;
        if (ptr_row >= min_row && ptr_row <= max_row && ptr_col >= min_col &&
ptr_col <= max_col)
        {
            *(roi_ptr+i) = roi_data.dataptr->getDataValue()[copydata_index];
            copydata_index++;
        }
    }
    cout << "After revision in ROI, ROI_final = " << endl;
    cout << *this;
    return *this;
}
```

◉ For user convenience, we overload the operator $*$ for several matrix multiplication algorithms rather than directly set them as member functions. However, this can also cause inconvenience because the same parameter of overloading operator $*$ will cause redifinition. Hence, each time we run the program, we need to comment out the other overloading for $*$. In this class, we overload the operator $*$ with the following algorithms: (more detailed analysis will be given in related test cases in $Part\ \#3$)

① general $i-k-j$ loop (in $Project\ \#2$, we prove that $i-k-j$ is the most efficient order to compute matrix multiplication for general $O(n^3)$ algorithms)

② $X86\ SSE$ optimization ($\_\_m128$: one loop, four calculations)

③ $ARM\ Neon$ optimization ($float32\times4\_t$: one loop, four calculations)

Remark:

✩ To meet the requirement, this matrix class should contain a member variable called $channel$. If $channel = 3$, this means that each data of the matrix is actually a $3-dim$ vector, just like 3 sheets in $Excel$. For convenience, we define the normal matrix computation $(+,-,*)$ as corresponding channel operations, $i.e.\ \ channel\ k + channel\ k,\ \ k = 1,2,3.$

✩ Compared with $Project\ \#3$, the programming language here is $C++$ instead of C. Hence, for operator overloading and member functions (friend functions), if we can return by reference or pass by reference, we avoid using objects. In this case, the program can be more efficient and also save more space (avoid copying objects).

✩ The pointer member variable which points to the matrix data is suggested to be $1-dim$ rather than $2-dim$. This is because $2-dim$ pointer usage needs to request memory twice, which may be discontinuous, while $1-dim$ case only needs once, avoiding this risk.

Part 2 – Code

There are **6** source code files: **data.hpp, data.cpp, mat.hpp, mat.cpp, mat_arm.cpp, main.cpp**. Since the code is relatively long, we do not show the whole code in this part (The complete code is attached in blackboard). Instead, we just show some key parts and give necessary explanations to describe the idea analyzed in $Part\ 1$.

◉ The design of the class:: $Data$ is shown in $Part\ 1$, so here we just show part of the definition of the member functions of class:: $Data$ in **data.cpp**:

```
Data::Data(size_t length)
{
    this->dataval = new myType[length]{};
    this->ptr_count = 1;
}

Data::~Data()
{
    delete[] this->dataval; // 清理矩阵的数据
    this->dataval = NULL;
}

myType * Data::getDataValue()
{
    return this->dataval;
}

void Data::addPtrCount()
{
    this->ptr_count++;
}

void Data::minusPtrCount()
{
    this->ptr_count--;
}

int Data::getPtrCount()
{
    return ptr_count;
}
```

① The destructor here is to delete the space of matrix data. Here the pointer $dataval$ points to the matrix data. We use $delete\ [\,]$ here to meet with $new\ [\,]$.

② The member variable ptr_count, as mentioned in $Part\ 1$, is to count the number of pointers which point to the space of the data of this matrix.

③ Since we need to change or get the value of ptr_count outside the class:: $Data$, we should write member functions: $addPtrCount(), minusPtrCount(), getPtrCount()$.

④ Surely, when overloading the operator $(+,-,*)$ in class:: Mat, we need to operate the matrix data. Hence, the member function $getDataValue()$ should be included.

◉ The design of Class:: Mat is shown as follows: (part of the code **mat.hpp**)

```
class Mat
{
private:
    int row;
    int col;
    int size;
    int channel;
    Data * dataptr;
    static int count;
    int mat_num;

public:
    Mat();
    Mat(int row, int col, int channel);
    Mat(int row, int col, const myType * data, int channel);
    Mat(const Mat & mat); // soft copy
    ~Mat();

    int getrownum() { return row;}
    int getcolnum() { return col;}

    bool operator == (const Mat & mat_cmp); // 比较矩阵元素
    Mat & operator = (const Mat & mat_copy); // 矩阵复制
    Mat operator + (const Mat & mat); // 矩阵加法
    Mat operator - (const Mat & mat); // 矩阵减法
    Mat operator * (const Mat & mat); // 矩阵乘法
    Mat operator ~ (); // 矩阵转置
    Mat & submat (int a, int b, string fileName); // ROI修改特定区域的矩阵数据

    friend Mat operator * (myType a, const Mat & rmat); // 矩阵左数乘
    friend Mat operator * (const Mat & lmat, myType a); // 矩阵右数乘

    friend ostream & operator << (ostream & os, const Mat & mat); // 输出矩阵到屏幕
    friend ifstream & operator >> (ifstream & ifs, const Mat & mat); // 从文件读取
矩阵
    friend ofstream & operator << (ofstream & ofs, const Mat & mat); // 输出矩阵到
文件
};

int getColNum(string fileName);
int getRowNum(string fileName);
```

① The member variables contains the number of row and column of the matrix: row, col. The definition of some member functions will use $row * col$ as judging conditions, so we also set the member variable $size$. $channel$ is already explained in the **remark** part in $Part\ 1$. $count$ is used to manage the matrices existing in the class already and also arrive at the end of its life cycle. It is **static** variable, we define it in **mat.cpp** because several code files will include **mat.hpp**. mat_num is designed to check the data of which matrix is deleted, and this important property will be tested in $Part\ 3$.

② The design of constructors, destructors and operator overloading functions are already explained in $Part\ 1$. Here we will explain the motivation why we design another class:: $Data$ based on $Part\ 1$. Also, we will analyze the design of destructor more carefully.

(i) **the motivation that we design another class::** $Data$:

◆ Since we overload the assignment operator $=$ by making two matrices share the same space of data (**soft copy**), we must consider how to delete this space just once, avoiding the problem of **double free**. At first, we want to declare the pointer pointing to matrix data as **smart pointer: shared_ptr**. However, although **shared_ptr** allows many pointers to point to the same space and can delete the space just once automatically, it must be defined to point to the new memory rather than the memory of data which has already existed. This is problematic.

◆ After that, we try to borrow the idea of **shared_ptr** rather than directly use it. We try to add a member variable of class:: Mat to count the number of pointers pointing to the space of matrix data. If there are other pointers pointing to this space with assigning operators or copy constructors called, the count will add 1, and if some pointers arrives at the end of their life cycle, the count will minus 1 after they are deleted. The memory of matrix data will be released if the count is 1.

◆ The idea is feasible. The problem is that the parameter of the assignment operator overloading function and the copy constructor are limited by the keyword **const**, meaning that we cannot change the value of the member variable. In this case, we design another class:: $Data$, and set the variable ptr_count which counts the number of pointers pointing to the space of matrix data as the member variable of class:: $Data$. Following this idea, the data type of the pointer member variable in class:: Mat will be changed from $myType$ to $Data$, and the pointer $dataval$ pointing to the matrix data is transfered into class:: $Data$.

(ii) **the design of destructor in class::** Mat**:**

```
Mat::~Mat()
{
    cout << "~Mat(), Mat # " << mat_num << endl;
    count--;
    if (this->dataptr == NULL)
    {
        cout << "Mat # " << mat_num << " data removed" << endl;
    }
    if (this->dataptr != NULL && this->dataptr->getPtrCount() == 1)
    {
        cout << "Mat # " << mat_num << " data removed" << endl;
        delete this->dataptr;
        this->dataptr = NULL;
    }
    if (this->dataptr != NULL && this->dataptr->getPtrCount() > 1)
    {
        cout << "Mat # " << mat_num << " ptr removed" << endl;
        this->dataptr->minusPtrCount();
    }
}
```

The basic idea follows the explanation in the above part (i). There are several work the destructor should do:

▪ update the number of matrix existing in the class:: Mat

▪ delete the space of matrix data if the count of pointers is 1

▪ just delete the pointers if the count of pointers is greater than 1

▪ after releasing the space of matrix data, delete the last pointer

To test the memory management of **soft copy**, we add the output of each case, and this will be shown in $Part\ 3$.

③ We do not overload the input operator $>>$ onto screen. This is because what we want to deal with is large scale matrix, which is not convenient to be input from keyboard onto screen. We overload the output operator $<<$ onto screen because we want to make easy test cases to test our program, as you will see in $Part\ 3$.

Part 3 – Result & Verification

In this part, we will make test cases for the following purposes:

$\star$ $Test\ case\ \#1$: test the member variable $channel = 2$, and for convenience, the remaining test cases are tested with $channel = 1$

$\star$ $Test\ case\ \#2$: test the 3 constructors for matrix creation

$\star$ $Test\ case\ \#3$: test the copy constructor for memory management

$\star$ $Test\ case\ \#4$: test the destructor and the assignment operator $=$ overloading function for memory management (**soft copy**)

$\star$ $Test\ case\ \#5$: test the overloading functions for general matrix operators $(+, -, *, T, ==)$ and files iostream operators $(<<, >>)$

$\star$ $Test\ case\ \#6$: test the member functions for ROI with **soft copy**

$\star$ $Test\ case\ \#7$: test the overloading functions of operator $*$ for general $i-k-j$ loop matrix multiplication in different platforms $(X86,\ ARM)$

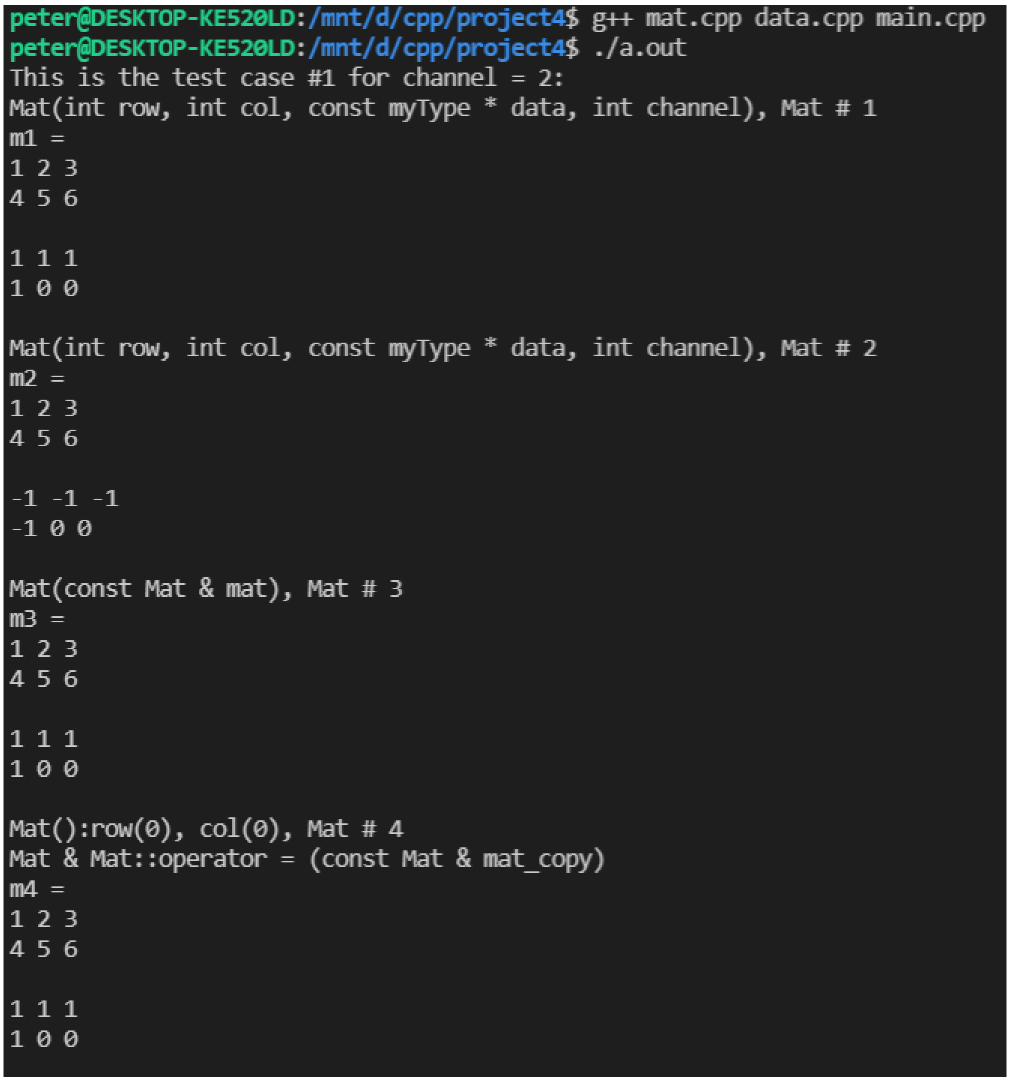
$\star$ $Test\ case\ \#8$: test the overloading functions of operator $*$ for matrix multiplication with optimization applied $(X86\ SSE,\ ARM\ Neon)$

For convenience, we just test the data type $float$.

```
#define myType float
```

Test case #1: (channel = 2)

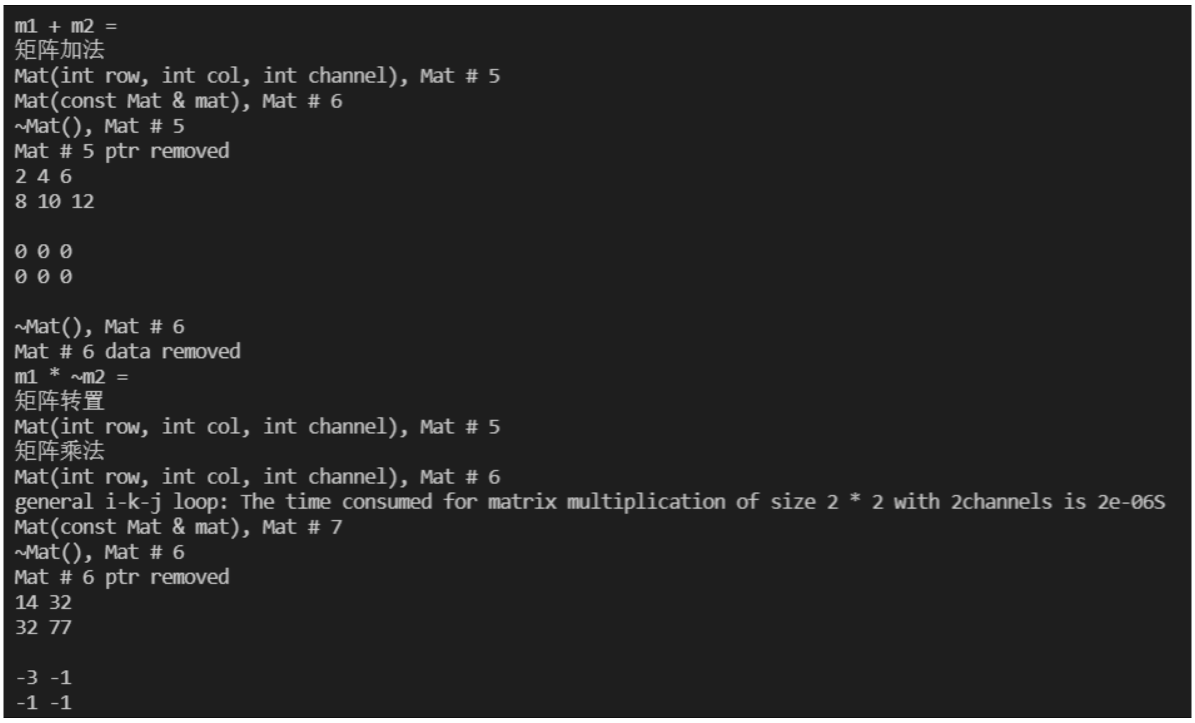
```
cout << "This is the test case #1 for channel = 2: " << endl;
myType m1_values[12] = {1.f,2.f,3.f,4.f,5.f,6.f,1.f,1.f,1.f,1.f};
Mat m1(2,3,m1_values,2);
cout << "m1 = " << endl;
cout << m1;

myType m2_values[12] = {1.f,2.f,3.f,4.f,5.f,6.f,-1.f,-1.f,-1.f,-1.f};
Mat m2(2,3,m2_values,2);
cout << "m2 = " << endl;
cout << m2;

Mat m3 = m1;
cout << "m3 = " << endl;
cout << m3;

Mat m4;
m4 = m1;
cout << "m4 = " << endl;
cout << m4;

cout << "m1 + m2 = " << endl;
cout << m1 + m2;

cout << "m1 * ~m2 = " << endl;
cout << m1 * ~m2;
```



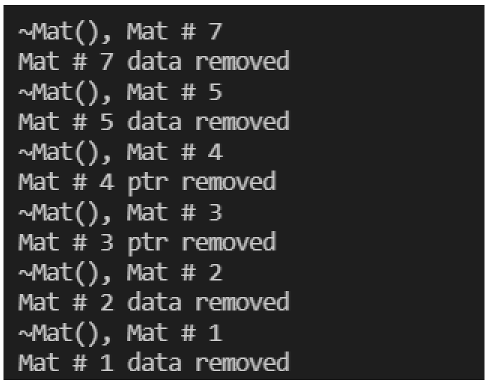
```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #1 for channel = 2:
Mat(int row, int col, const myType * data, int channel), Mat # 1
m1 =
1 2 3
4 5 6

1 1 1
1 0 0

Mat(int row, int col, const myType * data, int channel), Mat # 2
m2 =
1 2 3
4 5 6

-1 -1 -1
-1 0 0

Mat(const Mat & mat), Mat # 3
m3 =
1 2 3
4 5 6

1 1 1
1 0 0

Mat():row(0), col(0), Mat # 4
Mat & Mat::operator = (const Mat & mat_copy)
m4 =
1 2 3
4 5 6

1 1 1
1 0 0
```

① As you can see, there are two size $2 * 3$ matrices in one class:: Mat object m_1. It is equivalent that one element of matrix m_1 stores two float numbers.

② The constructors, copy constructor, and the overloading function of assignment operator $=$ work for $channel = 2$.



```
m1 + m2 =
矩阵加法
Mat(int row, int col, int channel), Mat # 5
Mat(const Mat & mat), Mat # 6
~Mat(), Mat # 5
Mat # 5 ptr removed
2 4 6
8 10 12

0 0 0
0 0 0

~Mat(), Mat # 6
Mat # 6 data removed
m1 * ~m2 =
矩阵转置
Mat(int row, int col, int channel), Mat # 5
矩阵乘法
Mat(int row, int col, int channel), Mat # 6
general i-k-j loop: The time consumed for matrix multiplication of size 2 * 2 with 2channels is 2e-06S
Mat(const Mat & mat), Mat # 7
~Mat(), Mat # 6
Mat # 6 ptr removed
14 32
32 77

-3 -1
-1 -1
```


```
~Mat(), Mat # 7
Mat # 7 data removed
~Mat(), Mat # 5
Mat # 5 data removed
~Mat(), Mat # 4
Mat # 4 ptr removed
~Mat(), Mat # 3
Mat # 3 ptr removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

③ Here we define the operation $(+, -, *)$ for matrices **channel-by-channel**. As you can see, the addition and the multiplication of matrices work in this way, $i.e.$

$$m_1[1] = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad m_1[2] = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

$$m_2[1] = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad m_2[2] = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 0 & 0 \end{pmatrix}$$

$$m_1 + m_2 = (m_1[1] + m_2[1],\ m_1[2] + m_2[2])$$

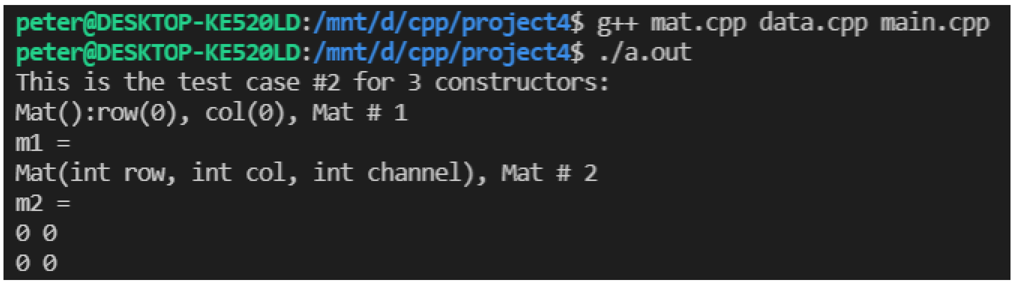
$$m_1 * m_2^T = (m_1[1] * m_2[1]^T,\ m_1[2] * m_2[2]^T)$$

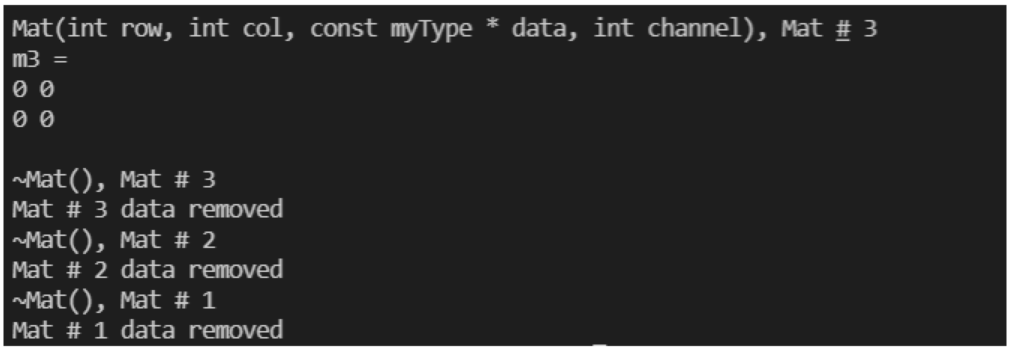
You can easily check that the result is consistent with that in the above screenshot.

④ The memory management is successful. We call copy constructor to copy m_1 $(Mat\ \#1)$ to m_3 $(Mat\ \#3)$ **softly**, and call overloading function of assignment operator $=$ to assign m_1 $(Mat\ \#1)$ to m_4 $(Mat\ \#4)$ softly. Hence, there are 3 pointers pointing to the matrix data of m_1. As you can see, we just delete the point of m_3 and m_4, and finally release the space of the data of m_1.

Test case #2: (3 constructors)

```
cout << "This is the test case #2 for 3 constructors: " << endl;
Mat m1;
cout << "m1 = " << endl;
cout << m1;
Mat m2(2,2,1);
cout << "m2 = " << endl;
cout << m2;
myType * values = new myType[6]();
Mat m3(2,2,values,1);
cout << "m3 = " << endl;
cout << m3;
```


```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #2 for 3 constructors:
Mat():row(0), col(0), Mat # 1
m1 =
Mat(int row, int col, int channel), Mat # 2
m2 =
0 0
0 0
```



```
Mat(int row, int col, const myType * data, int channel), Mat # 3
m3 =
0 0
0 0

~Mat(), Mat # 3
Mat # 3 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

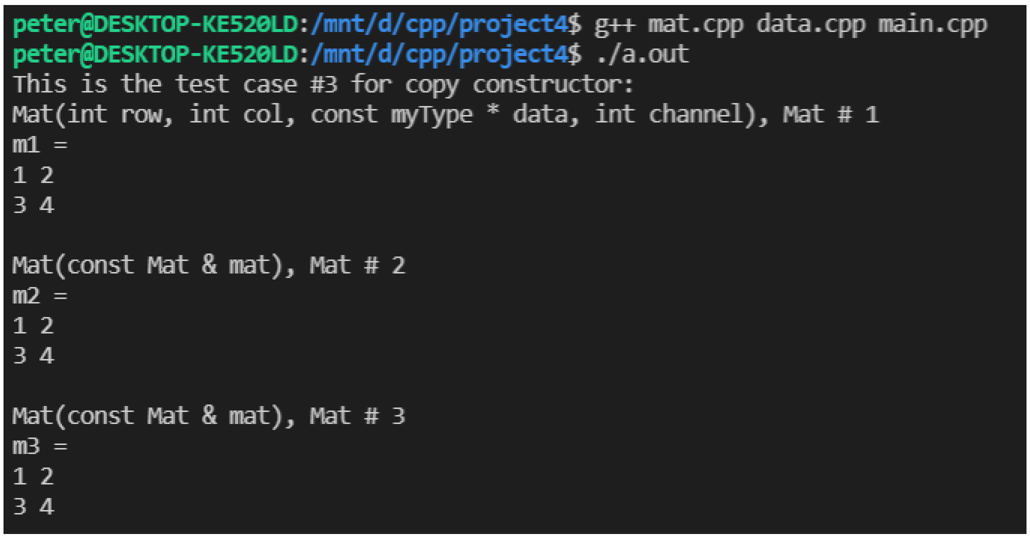
① As you can see, 3 constructors are called correspondingly to create the objects.

② m_1 does not have any data because we set the pointer $dataptr$ to be $NULL$.

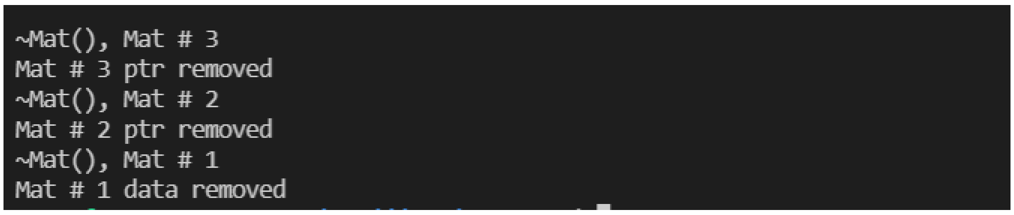
```
Mat::Mat():row(0), col(0)
{
    this->dataptr = NULL;
    count++;
    mat_num = count;
    cout << "Mat():row(0), col(0), Mat # " << mat_num << endl;
}
```

Test case #3: (copy constructor)

```
cout << "This is the test case #3 for copy constructor: " << endl;
myType values[4] = {1.f,2.f,3.f,4.f};
Mat m1(2,2,values,1);
cout << "m1 = " << endl;
cout << m1;
Mat m2(m1);
cout << "m2 = " << endl;
cout << m2;
Mat m3 = m1;
cout << "m3 = " << endl;
cout << m3;
```



```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #3 for copy constructor:
Mat(int row, int col, const myType * data, int channel), Mat # 1
m1 =
1 2
3 4

Mat(const Mat & mat), Mat # 2
m2 =
1 2
3 4

Mat(const Mat & mat), Mat # 3
m3 =
1 2
3 4
```

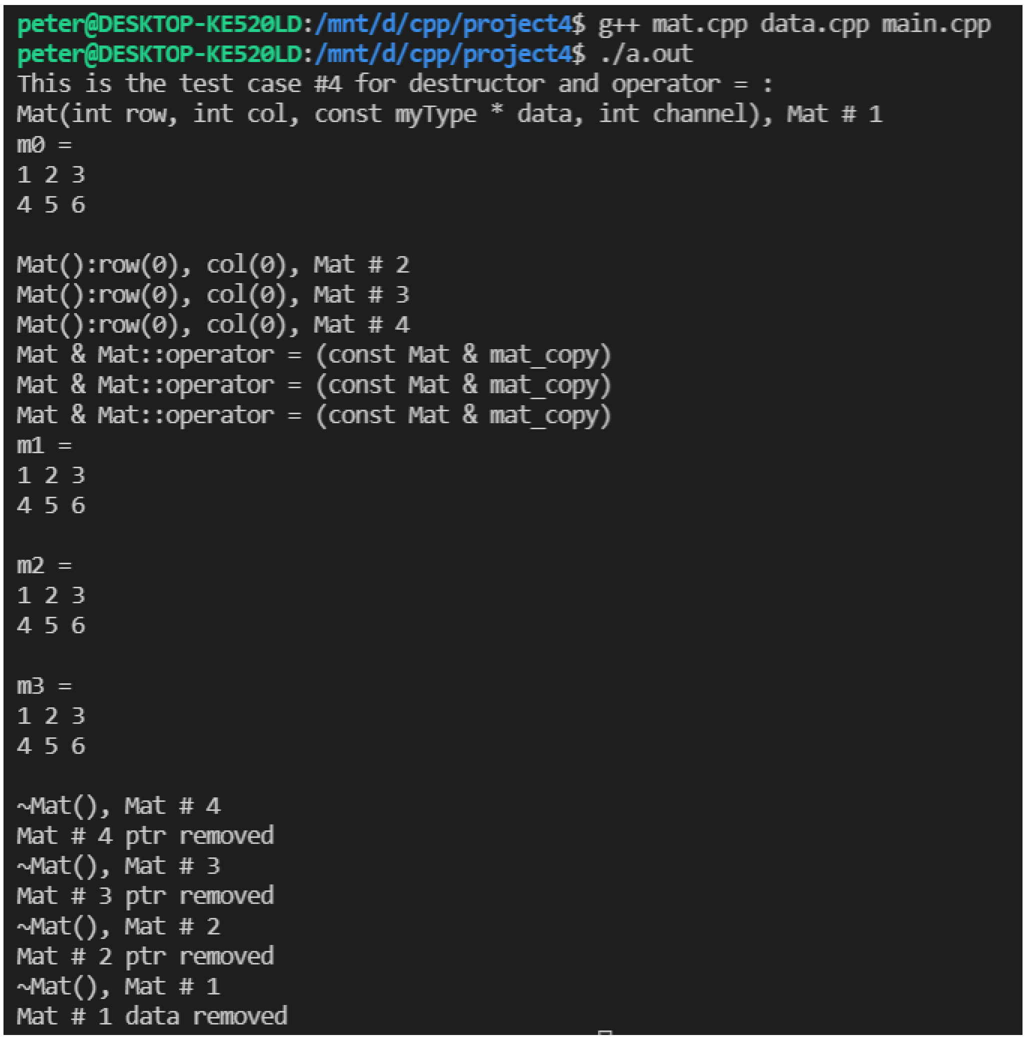


```
~Mat(), Mat # 3
Mat # 3 ptr removed
~Mat(), Mat # 2
Mat # 2 ptr removed
~Mat(), Mat # 1
Mat # 1 data removed
```

As you can see, the creation of matrices m_2, m_3 call the copy constructor in class:: Mat.

Test case #4: (destructor & assignment operator $=$ overloading function, soft copy)

```
cout << "This is the test case #4 for destructor and operator = : " << endl;
myType values[6] = {1.f,2.f,3.f,4.f,5.f,6.f};
Mat m0(2,3,values,1);
cout << "m0 = " << endl;
cout << m0;
Mat m1;
Mat m2;
Mat m3;
m1 = m2 = m3 = m0;
cout << "m1 = " << endl;
cout << m1;
cout << "m2 = " << endl;
cout << m2;
cout << "m3 = " << endl;
cout << m3;
```



```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #4 for destructor and operator = :
Mat(int row, int col, const myType * data, int channel), Mat # 1
m0 =
1 2 3
4 5 6

Mat():row(0), col(0), Mat # 2
Mat():row(0), col(0), Mat # 3
Mat():row(0), col(0), Mat # 4
Mat & Mat::operator = (const Mat & mat_copy)
Mat & Mat::operator = (const Mat & mat_copy)
Mat & Mat::operator = (const Mat & mat_copy)
m1 =
1 2 3
4 5 6

m2 =
1 2 3
4 5 6

m3 =
1 2 3
4 5 6

~Mat(), Mat # 4
Mat # 4 ptr removed
~Mat(), Mat # 3
Mat # 3 ptr removed
~Mat(), Mat # 2
Mat # 2 ptr removed
~Mat(), Mat # 1
Mat # 1 data removed
```

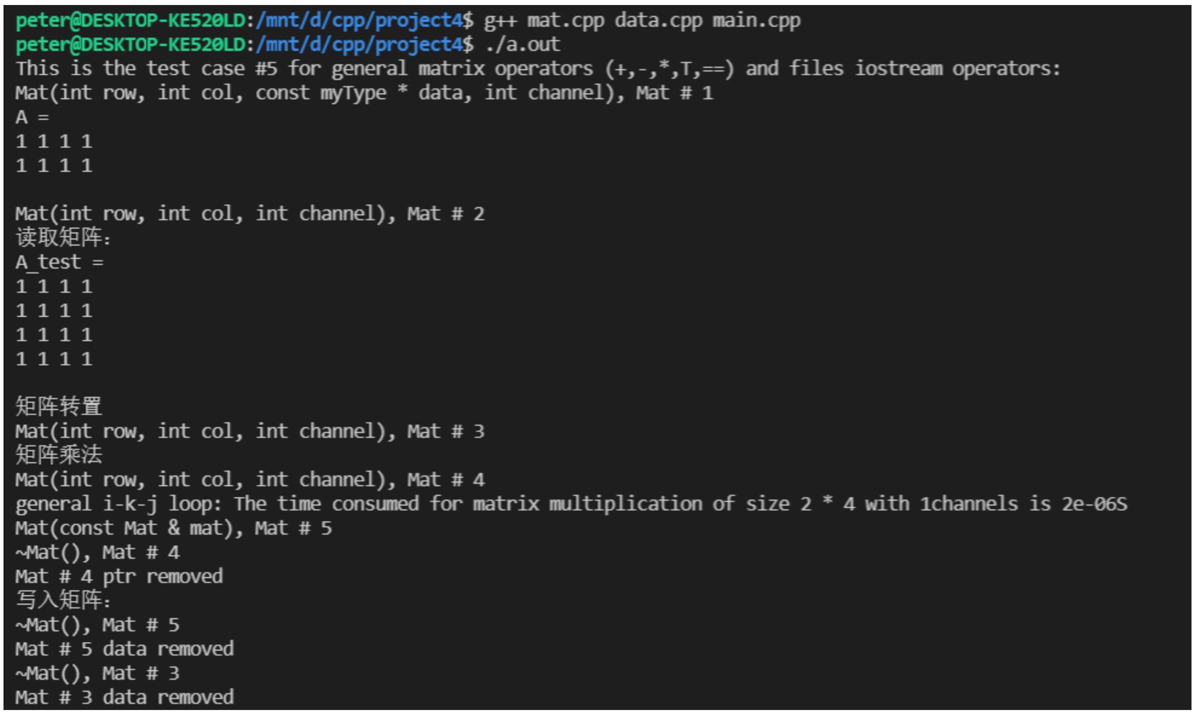
① As you can see, the code $m_1 = m_2 = m_3 = m_0$ calls the assignment operator $=$ overloading function 3 times.

② This actually achieves **soft copy** since 4 pointers point to the same space of data of matrix m_0.

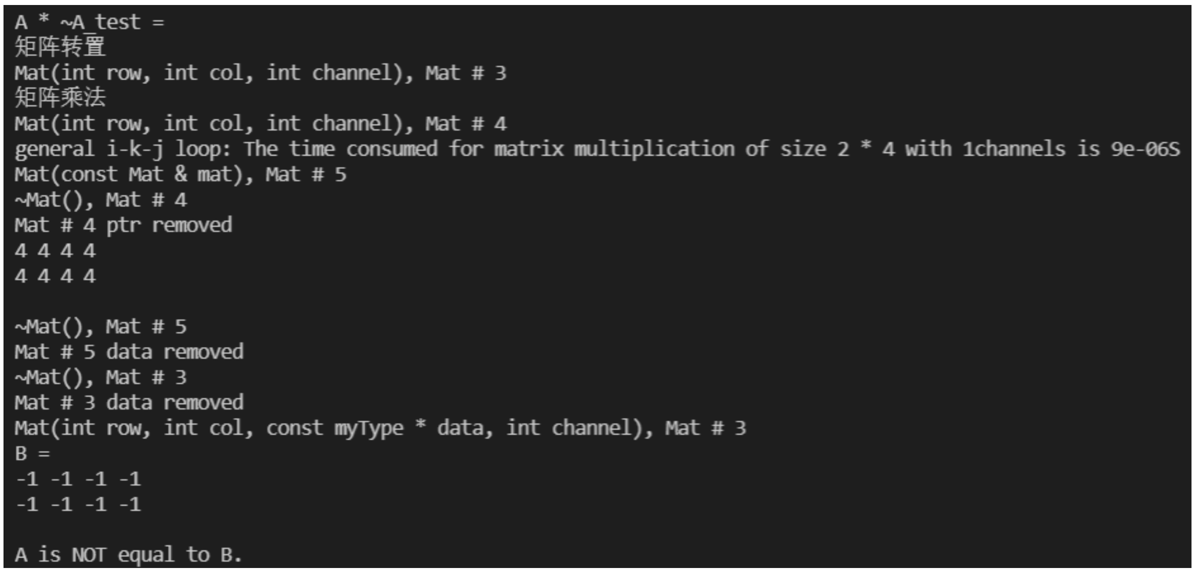
③ The memory management is successful since we remove the first 3 pointers and just delete the space of data **once**.

Test case #5: (general matrix operators $(+, -, *, T, ==)$ and files iostream operators $(<<, >>)$)

```
cout << "This is the test case #5 for general matrix operators (+,-,*,T,==) and
files iostream operators: " << endl;
myType A_Data[8] = {1,1,1,1,1,1,1,1};
Mat A = Mat(2,4,A_Data,1);
cout << "A = " << endl;
cout << A;

ifstream fin("ROI_original.txt");
int row = getRowNum("ROI_original.txt");
int col = getColNum("ROI_original.txt");
Mat A_test = Mat(row,col,1);
fin >> A_test;
fin.close();
cout << "A_test = " << endl;
cout << A_test;

ofstream ofile;
ofile.open("out_test_5.txt");
ofile << A * ~A_test;
ofile.close();
cout << "A * ~A_test = " << endl;
cout << A * ~A_test;

myType B_Data[8] = {-1,-1,-1,-1,-1,-1,-1,-1};
Mat B = Mat(2,4,B_Data,1);
cout << "B = " << endl;
cout << B;

bool AB = (A == B);
if (AB == 0)
{
    cout << "A is NOT equal to B." << endl;
}

Mat add = A + B;
cout << "add = A + B = " << endl;
cout << add;

cout << "A_test + B = " << endl;
cout << A_test + B; // invalid matrix addition

cout << "A - B = " << endl;
cout << A - B;

cout << "0.5f * A = " << endl;
cout << 0.5f * A;

cout << "A * 0.5f = " << endl;
cout << A * 0.5f;

cout << "A = " << endl;
cout << A;
```

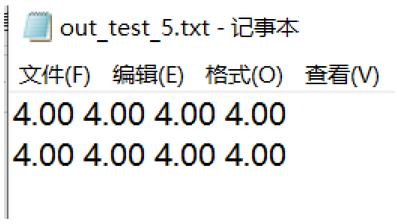


```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #5 for general matrix operators (+,-,*,T,==) and files iostream operators:
Mat(int row, int col, const myType * data, int channel), Mat # 1
A =
1 1 1 1
1 1 1 1

Mat(int row, int col, int channel), Mat # 2
读取矩阵:
A_test =
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

矩阵转置
Mat(int row, int col, int channel), Mat # 3
矩阵乘法
Mat(int row, int col, int channel), Mat # 4
general i-k-j loop: The time consumed for matrix multiplication of size 2 * 4 with 1channels is 2e-06S
Mat(const Mat & mat), Mat # 5
~Mat(), Mat # 4
Mat # 4 ptr removed
写入矩阵:
~Mat(), Mat # 5
Mat # 5 data removed
~Mat(), Mat # 3
Mat # 3 data removed
```



```
A * ~A_test =
矩阵转置
Mat(int row, int col, int channel), Mat # 3
矩阵乘法
Mat(int row, int col, int channel), Mat # 4
general i-k-j loop: The time consumed for matrix multiplication of size 2 * 4 with 1channels is 9e-06S
Mat(const Mat & mat), Mat # 5
~Mat(), Mat # 4
Mat # 4 ptr removed
4 4 4 4
4 4 4 4

~Mat(), Mat # 5
Mat # 5 data removed
~Mat(), Mat # 3
Mat # 3 data removed
Mat(int row, int col, const myType * data, int channel), Mat # 3
B =
-1 -1 -1 -1
-1 -1 -1 -1

A is NOT equal to B.
```

① A is a $2 * 4$ matrix with all entries to be 1. B is a $2 * 4$ matrix with all entries to be -1. A_test is loaded from the file **ROI_original.txt**, which is a $4 * 4$ matrix with all entries to be 1. As you can see, A_test is read from file and output onto screen successfully.

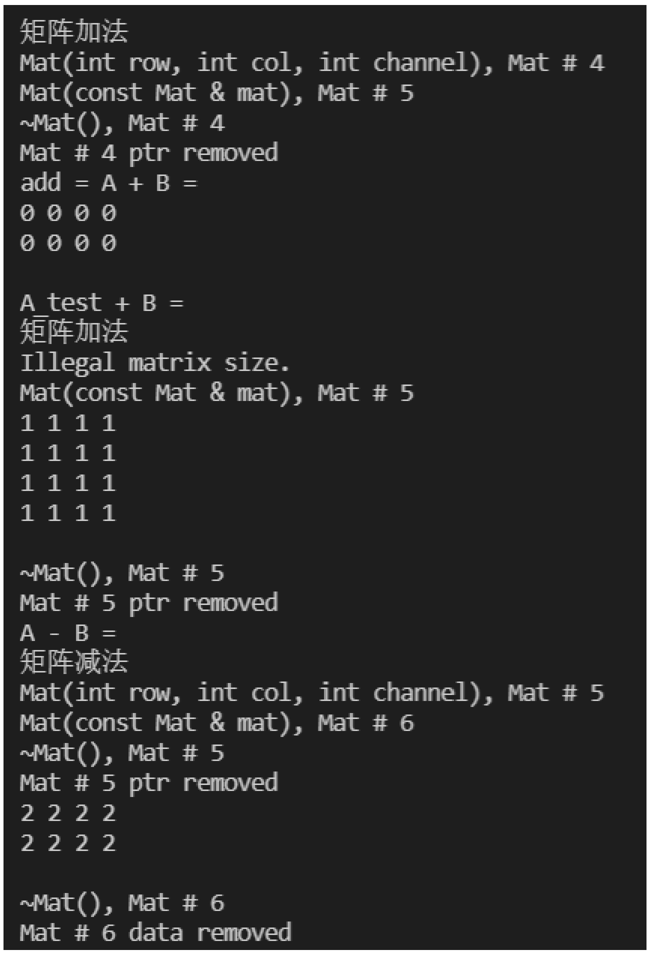


```
4.00 4.00 4.00 4.00
4.00 4.00 4.00 4.00
```

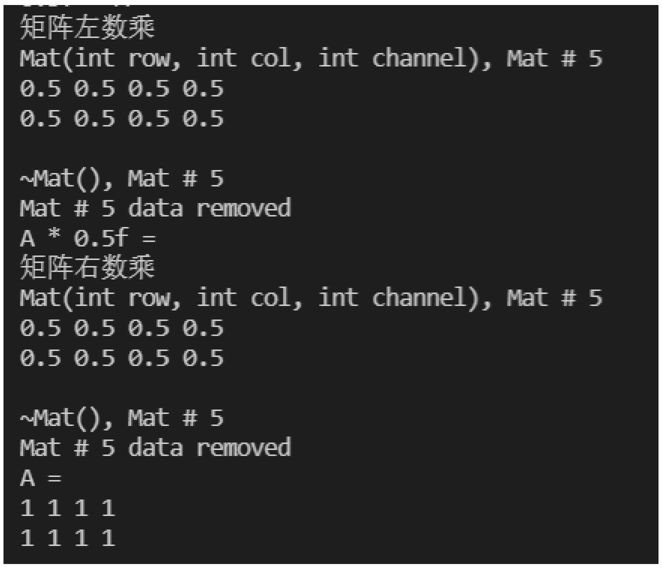
② We test the matrix multiplication with general $i - k - j$ loop for $A * A\_test^T$, which also tests the transpose of matrix.

③ We overload the operator $(+, -, *)$ with the object returned rather than reference. In the operator overloading function, we create a temporary matrix to store the result of operations of matrices and then return this temporary object. This is confirmed by the calling of constructors and destructor during the program.

④ We test the overloading function of operator $==$ by testing $A == B$. Since all entries of A are 1, while all entries of B are -1, they are NOT equal even if they have the same $row, col, size, channel$.

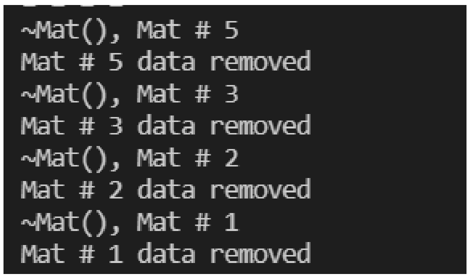


```
矩阵加法
Mat(int row, int col, int channel), Mat # 4
Mat(const Mat & mat), Mat # 5
~Mat(), Mat # 4
Mat # 4 ptr removed
add = A + B =
0 0 0 0
0 0 0 0

A_test + B =
矩阵加法
Illegal matrix size.
Mat(const Mat & mat), Mat # 5
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

~Mat(), Mat # 5
Mat # 5 ptr removed
A - B =
矩阵减法
Mat(int row, int col, int channel), Mat # 5
Mat(const Mat & mat), Mat # 6
~Mat(), Mat # 5
Mat # 5 ptr removed
2 2 2 2
2 2 2 2

~Mat(), Mat # 6
Mat # 6 data removed
```



```
矩阵左数乘
Mat(int row, int col, int channel), Mat # 5
0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5

~Mat(), Mat # 5
Mat # 5 data removed
A * 0.5f =
矩阵右数乘
Mat(int row, int col, int channel), Mat # 5
0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5

~Mat(), Mat # 5
Mat # 5 data removed
A =
1 1 1 1
1 1 1 1
```

④ No matter which operation matrix takes, the size must be right. For example, we test the case $A\_test + B$ to give the exception handling way (return the original matrix data). Moreover, other operations of matrices are tested as you can see.

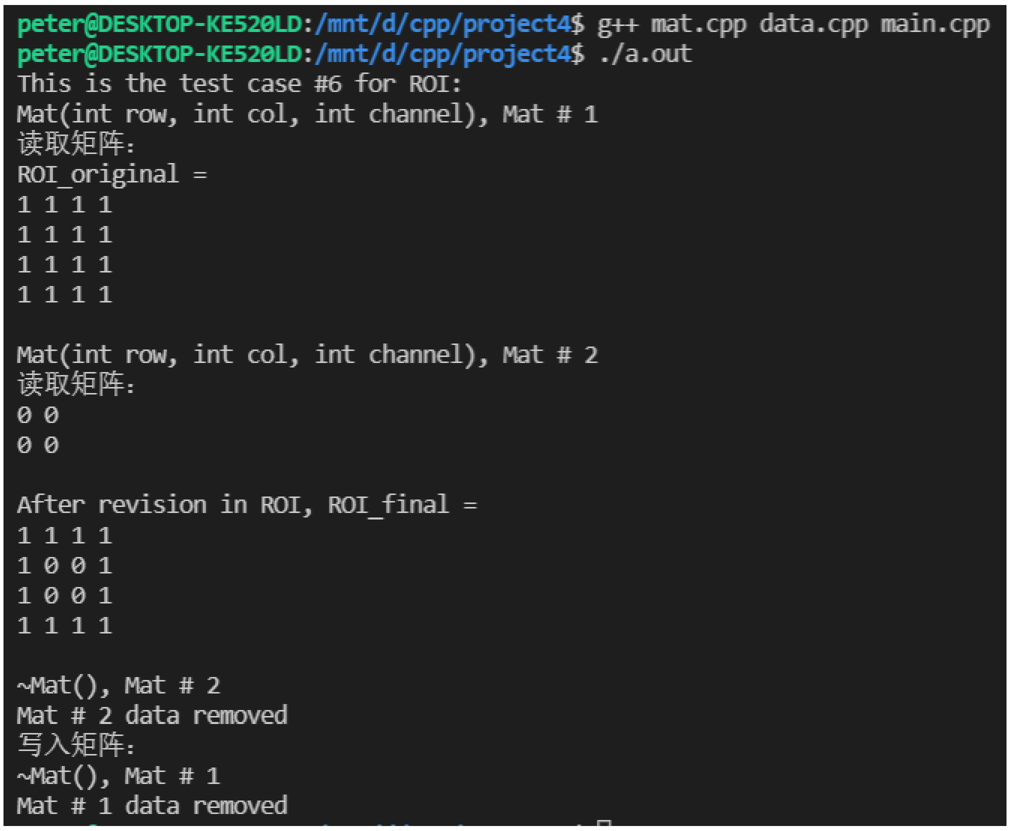


```
~Mat(), Mat # 5
Mat # 5 data removed
~Mat(), Mat # 3
Mat # 3 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

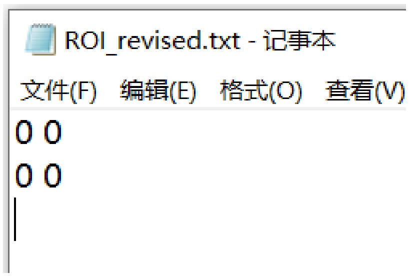
⑤ Since there are not assignments of matrices, the mechanism of smart pointers does not make an effect. The destructor directly release the space of matrix data.

Test case #6: (ROI with soft copy)

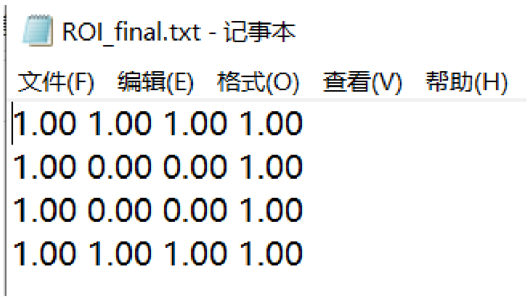
```
cout << "This is the test case #6 for ROI: " << endl;
ifstream fin("ROI_original.txt");
int ROI_row = getRowNum("ROI_original.txt");
int ROI_col = getColNum("ROI_original.txt");
Mat ROI_original = Mat(ROI_row, ROI_col,1);
fin >> ROI_original;
fin.close();
cout << "ROI_original = " << endl;
cout << ROI_original;

ROI_original.submat(6,9,"ROI_revised.txt");

ofstream ofile;
ofile.open("ROI_final.txt");
ofile << ROI_original;
ofile.close();
```



```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out
This is the test case #6 for ROI:
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
ROI_original =
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

Mat(int row, int col, int channel), Mat # 2
读取矩阵:
0 0
0 0

After revision in ROI, ROI_final =
1 1 1 1
1 0 0 1
1 0 0 1
1 1 1 1

~Mat(), Mat # 2
Mat # 2 data removed
写入矩阵:
~Mat(), Mat # 1
Mat # 1 data removed
```



```
0 0
0 0
```



```
1.00 1.00 1.00 1.00
1.00 0.00 0.00 1.00
1.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00
```

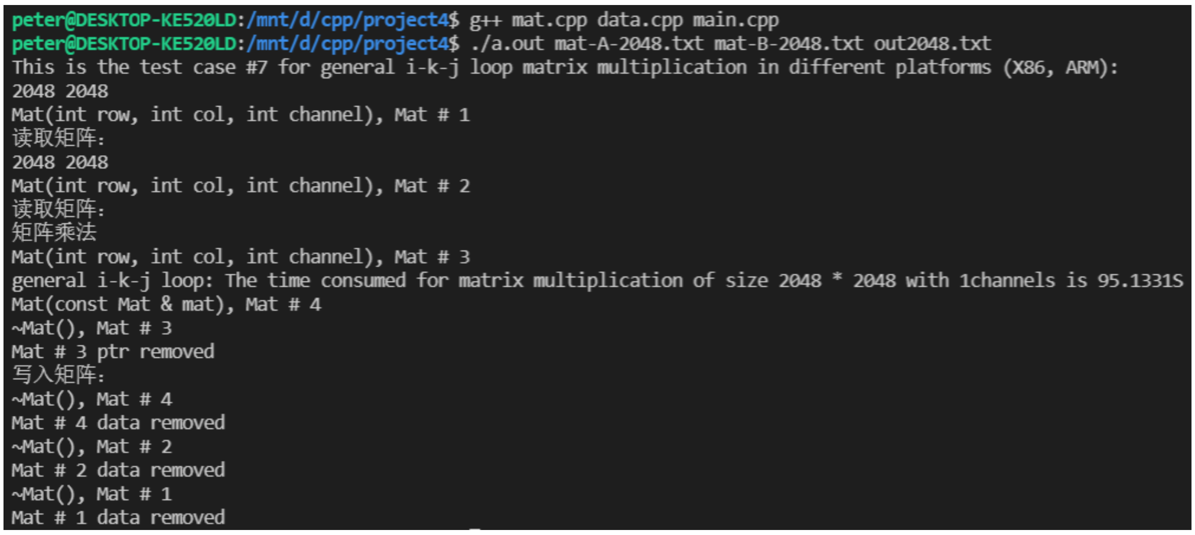
In this case, the ROI is the center $2 * 2$ submatrix (from index 5 to index 10 in $1 - dim$ data storage array). 2 given indices are 6 and 9, restricting the location of ROI. As you can see, the data in ROI is changed with revised data in **ROI_revised.txt**.

Test case #7: (general $i - k - j$ loop matrix multiplication in different platforms $(X86,\ ARM)$)

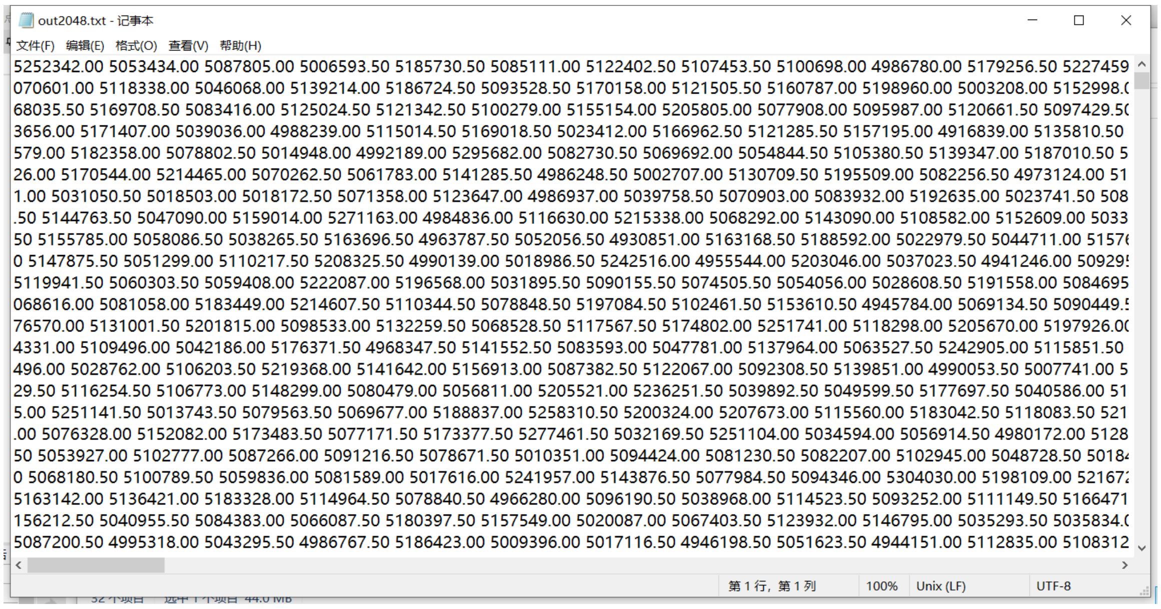
```
// Part of the code in mat.cpp
Mat Mat::operator * (const Mat & mat) // 矩阵乘法 general i-k-j loop
{
    cout << "矩阵乘法" << endl;
    if ((this->col != mat.row) || (this->channel != mat.channel))
    {
        cerr << "Illegal matrix size." << endl;
        return *this;
    }

    Mat result = Mat(this->row, mat.col, this->channel);
    result.size = this->row * mat.col;
    result.channel = this->channel;
    memset(result.dataptr->getDataValue(), 0, sizeof(myType) * result.row *
result.col * result.channel);

    clock_t start, end; // 计时起点和终点,只对计算部分计时
    start = clock(); // 计时开始
    for (int t = 0; t < result.channel; t++)
    {
        for (int i = 0; i < this->row; i++)
        {
            for (int k = 0; k < mat.row; k++)
            {
                for (int j = 0; j < mat.col; j++)
                {
                    result.dataptr->getDataValue()[t*result.size+i*mat.col+j] +=
this->dataptr->getDataValue()[t*this->size+i*this->col+k] * mat.dataptr-
>getDataValue()[t*mat.size+k*mat.col+j];
                }
            }
        }
    }
    end = clock();// 计时结束
    cout << "general i-k-j loop: The time consumed for matrix multiplication of
size " << this->row << " * " << mat.col
     << " with " << this->channel << "channels is " << double(end -
start)/CLOCKS_PER_SEC << "S" << endl;
    return result;
}
```

```
cout << "This is the test case #7 for general i-k-j loop matrix multiplication
in different platforms (X86, ARM): " << endl;
ifstream fin1(argv[1]);
int rowA1 = getRowNum(argv[1]);
int colA1 = getColNum(argv[1]);
cout << rowA1 << " " << colA1 << endl;
Mat A1 = Mat(rowA1,colA1,1);
fin1 >> A1;
fin1.close();

ifstream fin2(argv[2]);
int rowB1 = getRowNum(argv[2]);
int colB1 = getColNum(argv[2]);
cout << rowB1 << " " << colB1 << endl;
Mat B1 = Mat(rowB1,colB1,1);
fin2 >> B1;
fin2.close();

ofstream ofile;
ofile.open(argv[3]);
ofile << A1 * B1;
```

This is the test result on $X86$ platform:

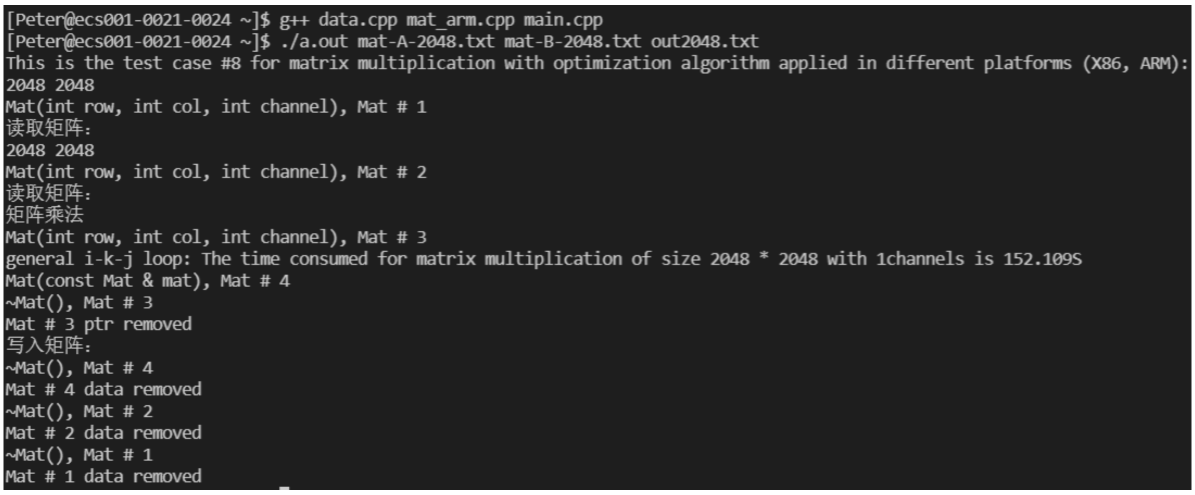


```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #7 for general i-k-j loop matrix multiplication in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
general i-k-j loop: The time consumed for matrix multiplication of size 2048 * 2048 with 1channels is 95.1331S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

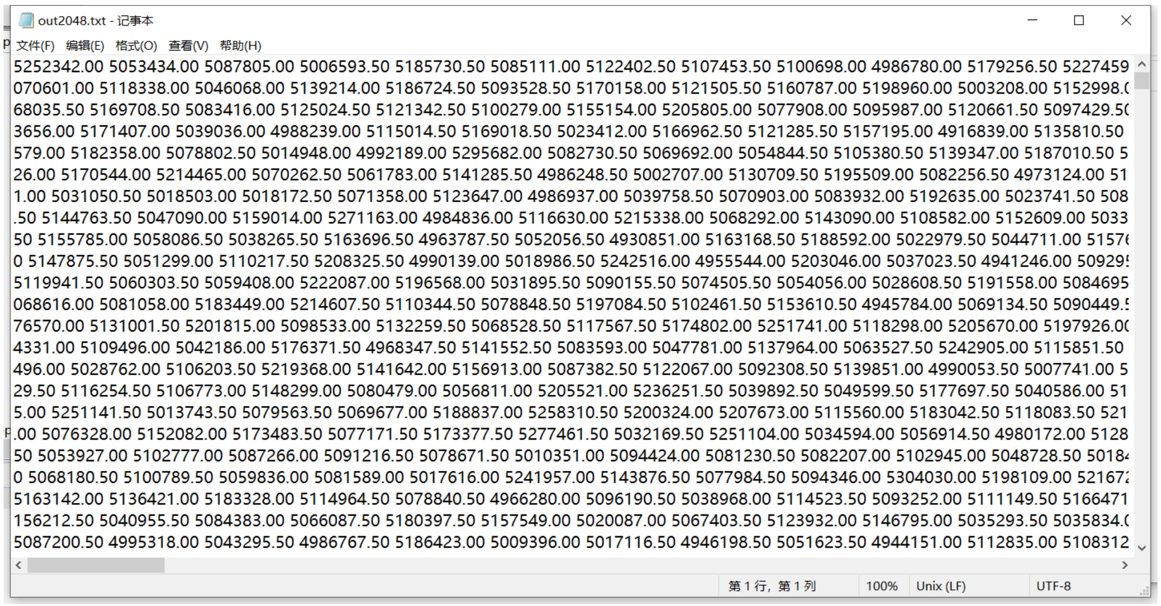


```
out2048.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
5252342.00 5053434.00 5087805.00 5006593.50 5185730.50 5085111.00 5122402.50 5107453.50 5100698.00 4986780.00 5179256.50 5227459
070601.00 5118338.00 5046068.00 5139214.00 5186724.50 5093528.50 5170158.00 5121505.50 5160787.00 5198960.00 5003208.00 5152998.
68035.50 5169708.50 5083416.00 5125024.50 5121342.50 5100279.00 5155154.00 5205805.00 5077908.00 5095987.00 5120661.50 5097429.5
3656.00 5171407.00 5039036.00 4988239.00 5115014.50 5169018.50 5023412.00 5166962.50 5121285.50 5157195.00 4916839.00 5135810.50
579.00 5182358.00 5078802.50 5014948.00 4992189.00 5295682.00 5082730.50 5069692.00 5054844.50 5105380.50 5139347.00 5187010.50 5
26.00 5170544.00 5214465.00 5070262.50 5061783.00 5141285.50 4986248.50 5002707.00 5130709.50 5195509.00 5082256.50 4973124.00 51
1.00 5031050.50 5018503.00 5018172.50 5071358.00 5123647.00 4986937.00 5039758.50 5070903.00 5083932.00 5192635.00 5023741.50 508
.50 5144763.50 5047090.00 5159014.00 5271163.00 4984836.00 5116630.00 5215338.00 5068292.00 5143090.00 5108582.00 5152609.00 5033
50 5155785.00 5058086.50 5038265.50 5163696.50 4963787.50 5052056.50 4930851.00 5163168.50 5188592.00 5022979.50 5044711.00 5157
0 5147875.50 5051299.00 5110217.50 5208325.50 4990139.00 5018986.50 5242516.00 4955544.00 5203046.00 5037023.50 4941246.00 50929
5119941.50 5060303.50 5059408.00 5222087.00 5196568.00 5031895.50 5090155.50 5074505.50 5054056.00 5028608.50 5191558.00 5084695
068616.00 5081058.00 5183449.00 5214607.50 5110344.50 5078848.50 5197084.50 5102461.50 5153610.50 4945784.00 5069134.50 5090449.
76570.00 5131001.50 5201815.00 5098533.00 5132259.50 5068528.50 5117567.50 5174802.00 5251741.00 5118298.00 5205670.00 5197926.0
4331.00 5109496.00 5042186.00 5176371.50 4968347.50 5141552.50 5083593.00 5047781.00 5137964.00 5063527.50 5242905.00 5115851.50
496.00 5028762.00 5106203.50 5219368.00 5141642.00 5156913.00 5087382.50 5122067.00 5092308.50 5139851.00 4990053.50 5007741.00 5
29.50 5116254.50 5106773.00 5148299.00 5080479.00 5056811.00 5205521.00 5236251.50 5039892.50 5049599.50 5177697.50 5040586.00 51
5.00 5251141.50 5013743.50 5079563.50 5069677.00 5188837.00 5258310.50 5200324.00 5207673.00 5115560.00 5183042.50 5118083.50 521
.00 5076328.00 5152082.00 5173483.50 5077171.50 5173377.50 5277461.50 5032169.50 5251104.00 5034594.00 5056914.50 4980172.00 5128
50 5053927.00 5102777.00 5087266.00 5091216.50 5078671.50 5010351.00 5094424.00 5081230.50 5082207.00 5102945.00 5048728.50 5018
0 5068180.50 5100789.50 5059836.00 5081589.00 5017616.00 5241957.00 5143876.50 5077984.50 5094346.00 5304030.00 5198109.00 52167
5163142.00 5136421.00 5183328.00 5114964.50 5078840.50 4966280.00 5096190.50 5038968.00 5114523.50 5093252.00 5111149.50 5166471
156212.50 5040955.50 5084383.00 5066087.50 5180397.50 5157549.00 5020087.00 5067403.50 5123932.00 5146795.00 5035293.50 5035834.
5087200.50 4995318.00 5043295.50 4986767.50 5186423.00 5009396.00 5017116.50 4946198.50 5051623.50 4944151.00 5112835.00 5108312
第 1 行，第 1 列    100%    Unix (LF)    UTF-8
```

This is the test result on ARM platform:



```
[Peter@ecs001-0021-0024 ~]$ g++ data.cpp mat_arm.cpp main.cpp
[Peter@ecs001-0021-0024 ~]$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #8 for matrix multiplication with optimization algorithm applied in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
general i-k-j loop: The time consumed for matrix multiplication of size 2048 * 2048 with 1channels is 152.109S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```



```
out2048.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
5252342.00 5053434.00 5087805.00 5006593.50 5185730.50 5085111.00 5122402.50 5107453.50 5100698.00 4986780.00 5179256.50 5227459
070601.00 5118338.00 5046068.00 5139214.00 5186724.50 5093528.50 5170158.00 5121505.50 5160787.00 5198960.00 5003208.00 5152998.
68035.50 5169708.50 5083416.00 5125024.50 5121342.50 5100279.00 5155154.00 5205805.00 5077908.00 5095987.00 5120661.50 5097429.5
3656.00 5171407.00 5039036.00 4988239.00 5115014.50 5169018.50 5023412.00 5166962.50 5121285.50 5157195.00 4916839.00 5135810.50
579.00 5182358.00 5078802.50 5014948.00 4992189.00 5295682.00 5082730.50 5069692.00 5054844.50 5105380.50 5139347.00 5187010.50 5
26.00 5170544.00 5214465.00 5070262.50 5061783.00 5141285.50 4986248.50 5002707.00 5130709.50 5195509.00 5082256.50 4973124.00 51
1.00 5031050.50 5018503.00 5018172.50 5071358.00 5123647.00 4986937.00 5039758.50 5070903.00 5083932.00 5192635.00 5023741.50 508
.50 5144763.50 5047090.00 5159014.00 5271163.00 4984836.00 5116630.00 5215338.00 5068292.00 5143090.00 5108582.00 5152609.00 5033
50 5155785.00 5058086.50 5038265.50 5163696.50 4963787.50 5052056.50 4930851.00 5163168.50 5188592.00 5022979.50 5044711.00 5157
0 5147875.50 5051299.00 5110217.50 5208325.50 4990139.00 5018986.50 5242516.00 4955544.00 5203046.00 5037023.50 4941246.00 50929
5119941.50 5060303.50 5059408.00 5222087.00 5196568.00 5031895.50 5090155.50 5074505.50 5054056.00 5028608.50 5191558.00 5084695
068616.00 5081058.00 5183449.00 5214607.50 5110344.50 5078848.50 5197084.50 5102461.50 5153610.50 4945784.00 5069134.50 5090449.
76570.00 5131001.50 5201815.00 5098533.00 5132259.50 5068528.50 5117567.50 5174802.00 5251741.00 5118298.00 5205670.00 5197926.0
4331.00 5109496.00 5042186.00 5176371.50 4968347.50 5141552.50 5083593.00 5047781.00 5137964.00 5063527.50 5242905.00 5115851.50
496.00 5028762.00 5106203.50 5219368.00 5141642.00 5156913.00 5087382.50 5122067.00 5092308.50 5139851.00 4990053.50 5007741.00 5
29.50 5116254.50 5106773.00 5148299.00 5080479.00 5056811.00 5205521.00 5236251.50 5039892.50 5049599.50 5177697.50 5040586.00 51
5.00 5251141.50 5013743.50 5079563.50 5069677.00 5188837.00 5258310.50 5200324.00 5207673.00 5115560.00 5183042.50 5118083.50 521
.00 5076328.00 5152082.00 5173483.50 5077171.50 5173377.50 5277461.50 5032169.50 5251104.00 5034594.00 5056914.50 4980172.00 5128
50 5053927.00 5102777.00 5087266.00 5091216.50 5078671.50 5010351.00 5094424.00 5081230.50 5082207.00 5102945.00 5048728.50 5018
0 5068180.50 5100789.50 5059836.00 5081589.00 5017616.00 5241957.00 5143876.50 5077984.50 5094346.00 5304030.00 5198109.00 52167
5163142.00 5136421.00 5183328.00 5114964.50 5078840.50 4966280.00 5096190.50 5038968.00 5114523.50 5093252.00 5111149.50 5166471
156212.50 5040955.50 5084383.00 5066087.50 5180397.50 5157549.00 5020087.00 5067403.50 5123932.00 5146795.00 5035293.50 5035834.
5087200.50 4995318.00 5043295.50 4986767.50 5186423.00 5009396.00 5017116.50 4946198.50 5051623.50 4944151.00 5112835.00 5108312
第 1 行，第 1 列    100%    Unix (LF)    UTF-8
```

① The results of $2048 * 2048$ matrix multiplication on $X86$ and ARM platform are the same.

② The only difference when the general $i-k-j$ loop matrix multiplication is overloaded for these two platforms is the computation speed. After several tests, we conclude that ARM is **slower** than $X86$ in general.

◉ Based on ①②, we can conclude that this matrix class with the general $i-k-j$ loop matrix multiplication overloaded can work in 2 different platforms $(X86,\ ARM)$.

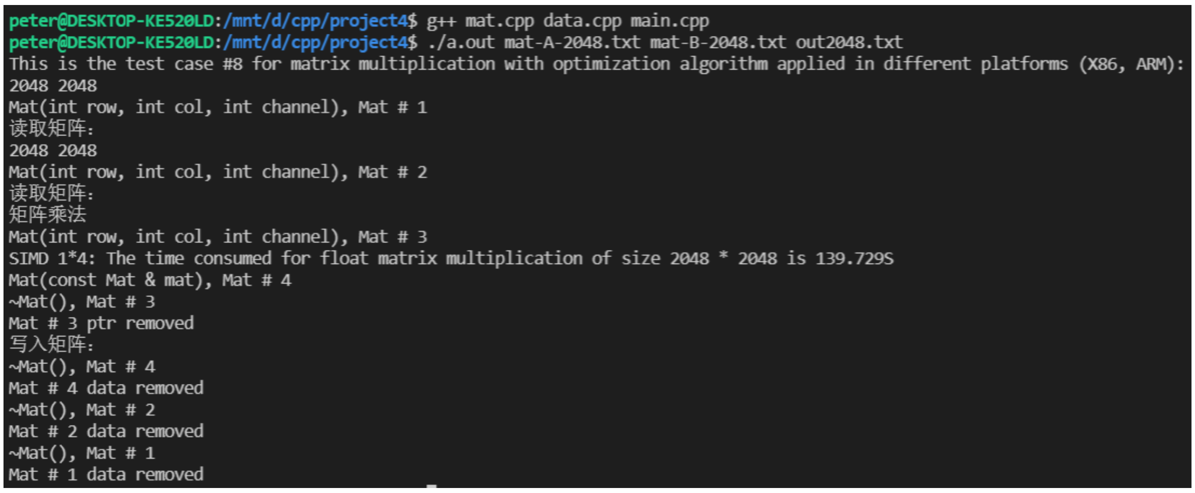
Test case #8: (matrix multiplication with optimization applied $(X86\ SSE,\ ARM\ Neon)$**)**

✔ This is the test result of matrix multiplication with $X86\ SSE$ $1*4$ applied:

```
// Part of code in mat_arm.cpp
Mat Mat::operator * (const Mat & mat) // 矩阵乘法，SIMD 1*4
{
    cout << "矩阵乘法" << endl;
    if (this->col != mat.row)
    {
        cerr << "Illegal matrix size." << endl;
        return *this;
    }

    Mat result = Mat(this->row, mat.col,1);
    result.size = this->row * mat.col;
    memset(result.dataptr->getDataValue(), 0, sizeof(myType) * result.row *
result.col);

    clock_t start, end; // 计时起点和终点,只对计算部分计时
    start = clock(); // 计时开始
    int len = this->col / 4;
    for( int j = 0; j < mat.col; j++)
    {
        for (int k = 0; k < len; k++)
        {
            __m128 m2 = _mm_set_ps(mat.dataptr->getDataValue()
[(4*k+3)*mat.col+j],mat.dataptr->getDataValue()[(4*k+2)*mat.col+j]
            ,mat.dataptr->getDataValue()[(4*k+1)*mat.col+j],mat.dataptr-
>getDataValue()[4*k*mat.col+j]);
            for (int i = 0; i < this->row; i++)
            {
                __m128 m1 = _mm_load_ps(&this->dataptr->getDataValue()[i*this-
>col + 4*k]);
                __m128 temp = _mm_mul_ps(m1, m2);
                myType buffer[4];
                _mm_store_ps(buffer,temp);
                result.dataptr->getDataValue()[i*result.col+j] +=
buffer[0]+buffer[1]+buffer[2]+buffer[3];
            }
        }
    }
    end = clock(); // 计时结束
    cout << "SIMD 1*4: The time consumed for float matrix multiplication of size
" << this->row << " * " << mat.col
     << " is " << double(end - start)/CLOCKS_PER_SEC << "S" << endl;

    return result;
}
```

```
cout << "This is the test case #8 for matrix multiplication with optimization
algorithm applied in different platforms (X86, ARM): " << endl;
ifstream fin1(argv[1]);
int rowA1 = getRowNum(argv[1]);
int colA1 = getColNum(argv[1]);
```

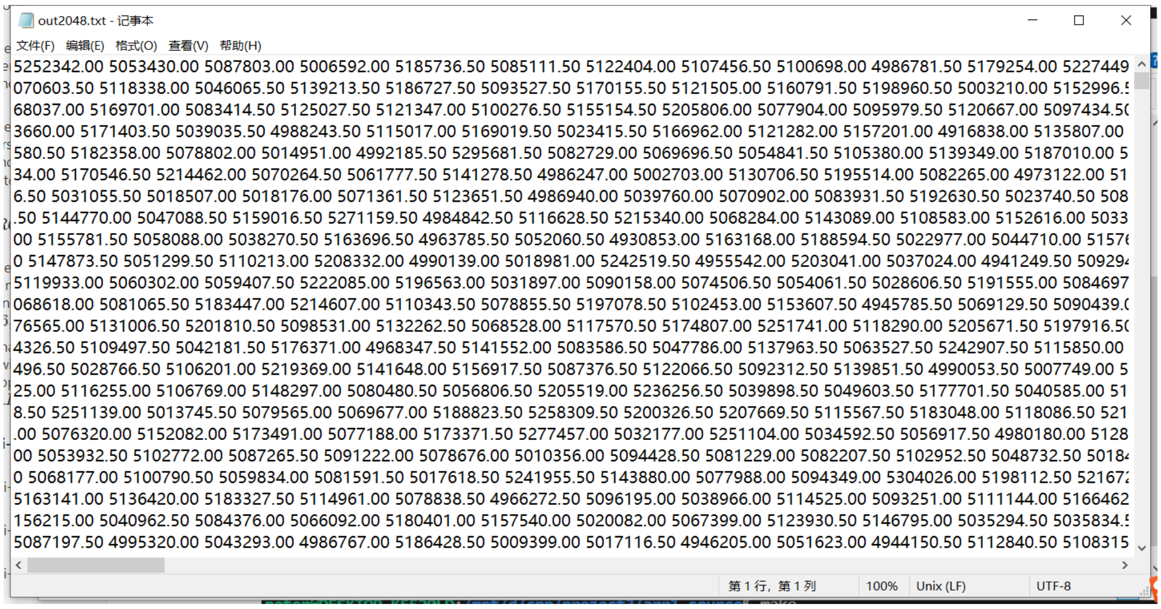
```
cout << rowA1 << " " << colA1 << endl;
Mat A1 = Mat(rowA1,colA1,1);
fin1 >> A1;
fin1.close();

ifstream fin2(argv[2]);
int rowB1 = getRowNum(argv[2]);
int colB1 = getColNum(argv[2]);
cout << rowB1 << " " << colB1 << endl;
Mat B1 = Mat(rowB1,colB1,1);
fin2 >> B1;
fin2.close();

ofstream ofile;
ofile.open(argv[3]);
ofile << A1 * B1;
```



```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #8 for matrix multiplication with optimization algorithm applied in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
SIMD 1*4: The time consumed for float matrix multiplication of size 2048 * 2048 is 139.729S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

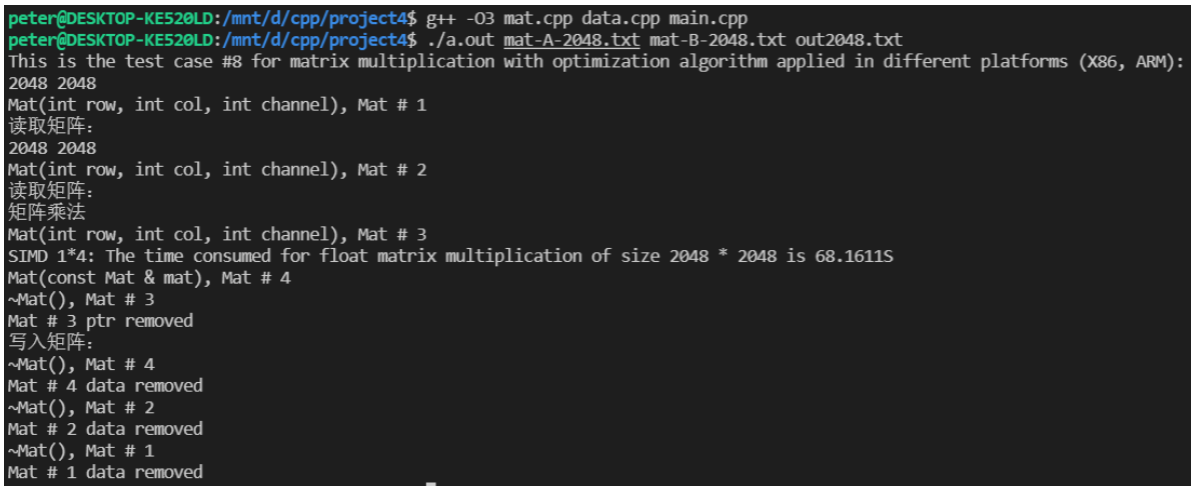
① Compared with the general $i-k-j$ loop $(95.1331s)$, $SSE\ 1*4$ is much **slower**. The possible explanation is that reordering the three loops can be more effcient than computing 4 terms in one loop. If you compare $SSE\ 1*4$ with the original order $i-j-k$, you will find that $SSE\ 1*4$ is much faster.

② Although the speed may not be as fast as the general $i-k-j$ loop, the **accuracy** of $SSE\ 1*4$ is better since in each loop we compute 4 terms and add them up, which **decreases the rounding error**. The result of the matrix multiplication by using $SSE,\ 1*4$ is shown bellow, you can compare it with that when the general $i-k-j$ loop applied.



```
5252342.00 5053430.00 5087803.00 5006592.00 5185736.50 5085111.50 5122404.00 5107456.50 5100698.00 4986781.50 5179254.00 5227449
070603.50 5118338.00 5046065.50 5139213.50 5186727.50 5093527.50 5170155.50 5121505.00 5160791.50 5198960.50 5003210.00 5152996.5
68037.00 5169701.00 5083414.50 5125027.50 5121347.00 5100276.50 5155154.50 5205806.00 5077904.00 5095979.50 5120667.00 5097434.50
3660.00 5171403.50 5039035.50 4988243.50 5115017.00 5169019.50 5023415.50 5166962.00 5121282.00 5157201.00 4916838.00 5135807.00
580.50 5182358.00 5078802.00 5014951.00 4992185.50 5295681.50 5082729.00 5069696.50 5054841.50 5105380.00 5139349.00 5187010.00 5
34.00 5170546.50 5214462.00 5070264.50 5061777.50 5141278.50 4986247.00 5002703.00 5130706.50 5195514.00 5082265.00 4973122.00 51
6.50 5031055.50 5018507.00 5018176.00 5071361.50 5123651.50 4986940.00 5039760.00 5070902.00 5083931.50 5192630.50 5023740.50 508
.50 5144770.00 5047088.50 5159016.50 5271159.50 4984842.50 5116628.50 5215340.00 5068284.00 5143089.00 5108583.00 5152616.00 5033
00 5155781.50 5058088.00 5038270.50 5163696.50 4963785.50 5052060.50 4930853.00 5163168.00 5188594.50 5022977.00 5044710.00 5157
0 5147873.50 5051299.50 5110213.00 5208332.00 4990139.00 5018981.00 5242519.50 4955542.00 5203041.00 5037024.00 4941249.50 50929
5119933.00 5060302.00 5059407.50 5222085.00 5196563.00 5031897.00 5090158.00 5074506.50 5054061.50 5028606.50 5191555.00 5084697
068618.00 5081065.50 5183447.00 5214607.00 5110343.50 5078855.50 5197078.50 5102453.00 5153607.50 4945785.50 5069129.50 5090439.0
76565.00 5131006.50 5201810.50 5098531.00 5132262.50 5068528.00 5117570.50 5174807.00 5251741.00 5118290.00 5205671.50 5197916.50
4326.50 5109497.50 5042181.50 5176371.00 4968347.50 5141552.00 5083586.50 5047786.00 5137963.50 5063527.50 5242907.50 5115850.00
496.50 5028766.50 5106201.00 5219369.00 5141648.00 5156917.50 5087376.50 5122066.50 5092312.50 5139851.50 4990053.50 5007749.00 5
25.00 5116255.00 5106769.00 5148297.00 5080480.50 5056806.50 5205519.00 5236256.50 5039898.50 5049603.50 5177701.50 5040585.00 51
8.50 5251139.00 5013745.50 5079565.00 5069677.00 5188823.50 5258309.50 5200326.50 5207669.50 5115567.50 5183048.00 5118086.50 521
.00 5076320.00 5152082.00 5173491.00 5077188.00 5173371.50 5277457.00 5032177.00 5251104.00 5034592.50 5056917.50 4980180.00 5128
00 5053932.50 5102772.00 5087265.50 5091222.00 5078676.00 5010356.00 5094428.50 5081229.00 5082207.50 5102952.50 5048732.50 5018
0 5068177.00 5100790.50 5059834.00 5081591.50 5017618.50 5241955.50 5143880.00 5077988.00 5094349.00 5304026.00 5198112.50 52167
5163141.00 5136420.00 5183327.50 5114961.00 5078838.50 4966272.50 5096195.00 5038966.00 5114525.00 5093251.00 5111144.00 5166462
156215.00 5040962.50 5084376.00 5066092.00 5180401.00 5157540.00 5020082.00 5067399.00 5123930.50 5146795.00 5035294.50 5035834.5
5087197.50 4995320.00 5043293.00 4986767.00 5186428.50 5009399.00 5017116.50 4946205.00 5051623.00 4944150.50 5112840.50 5108315
```

③ We can speed up the matrix multiplication by using $-O3$ to compile the code, and the result is shown as follows:



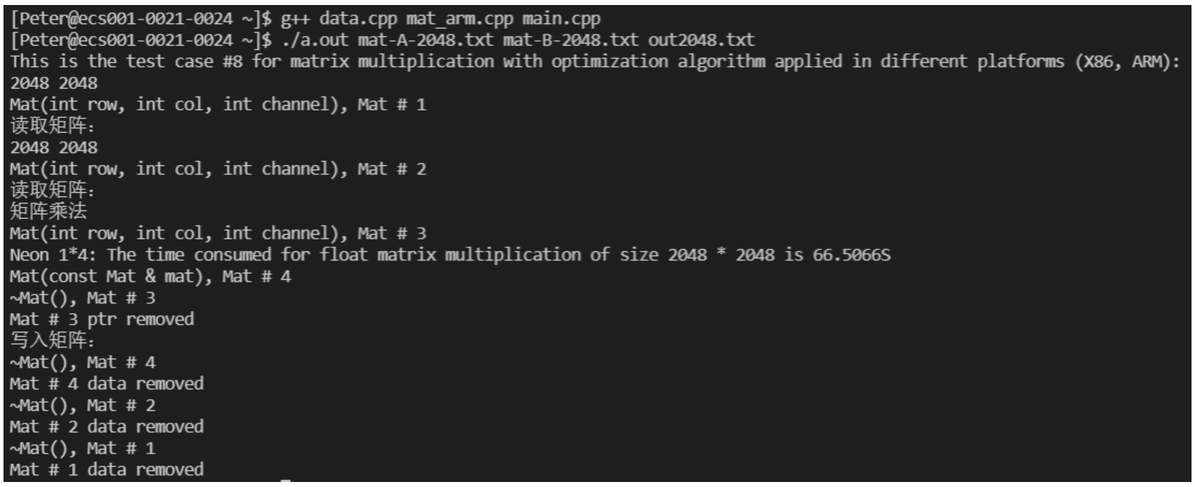
```
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ g++ -O3 mat.cpp data.cpp main.cpp
peter@DESKTOP-KE520LD:/mnt/d/cpp/project4$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #8 for matrix multiplication with optimization algorithm applied in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
SIMD 1*4: The time consumed for float matrix multiplication of size 2048 * 2048 is 68.1611S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

We can see that the computation is a lot faster.

✔ This is the test result of matrix multiplication with $ARM\ Neon\ 1*4$ applied:

```
Mat Mat::operator * (const Mat & mat) // 矩阵乘法，Neon 1*4
{
    cout << "矩阵乘法" << endl;
    if (this->col != mat.row)
    {
        cerr << "Illegal matrix size." << endl;
        return *this;
    }

    Mat result = Mat(this->row, mat.col,1);
    result.size = this->row * mat.col;
    memset(result.dataptr->getDataValue(), 0, sizeof(myType) * result.row *
result.col);

    clock_t start, end; // 计时起点和终点,只对计算部分计时
    start = clock(); // 计时开始
    int len = this->col / 4;
    for( int j = 0; j < mat.col; j++)
    {
        for (int k = 0; k < len; k++)
        {
            float temp[] = {mat.dataptr->getDataValue()
[4*k*mat.col+j],mat.dataptr->getDataValue()[(4*k+1)*mat.col+j],mat.dataptr-
>getDataValue()[(4*k+2)*mat.col+j],mat.dataptr->getDataValue()
[(4*k+3)*mat.col+j]};
            float32x4_t m2 = vld1q_f32(temp);
            for (int i = 0; i < this->row; i++)
            {
                float32x4_t m1 = vld1q_f32(&this->dataptr->getDataValue()
[i*this->col + 4*k]);
                float32x4_t temp = vmulq_f32(m1, m2);
                myType buffer[4];
                vst1q_f32(buffer,temp);
                result.dataptr->getDataValue()[i*result.col+j] +=
buffer[0]+buffer[1]+buffer[2]+buffer[3];
            }
        }
```

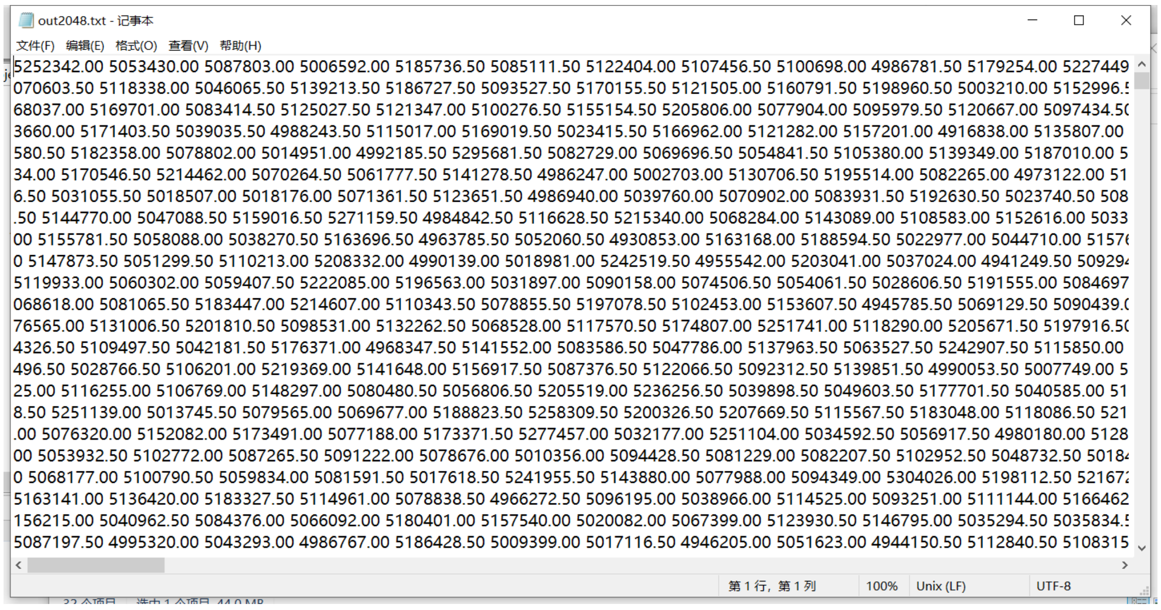
```cpp
    }
    end = clock(); // 计时结束
    cout << "Neon 1*4: The time consumed for float matrix multiplication of size
" << this->row << " * " << mat.col
     << " is " << double(end - start)/CLOCKS_PER_SEC << "S" << endl;

    return result;
}
```



```
[Peter@ecs001-0021-0024 ~]$ g++ data.cpp mat_arm.cpp main.cpp
[Peter@ecs001-0021-0024 ~]$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #8 for matrix multiplication with optimization algorithm applied in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
Neon 1*4: The time consumed for float matrix multiplication of size 2048 * 2048 is 66.5066S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

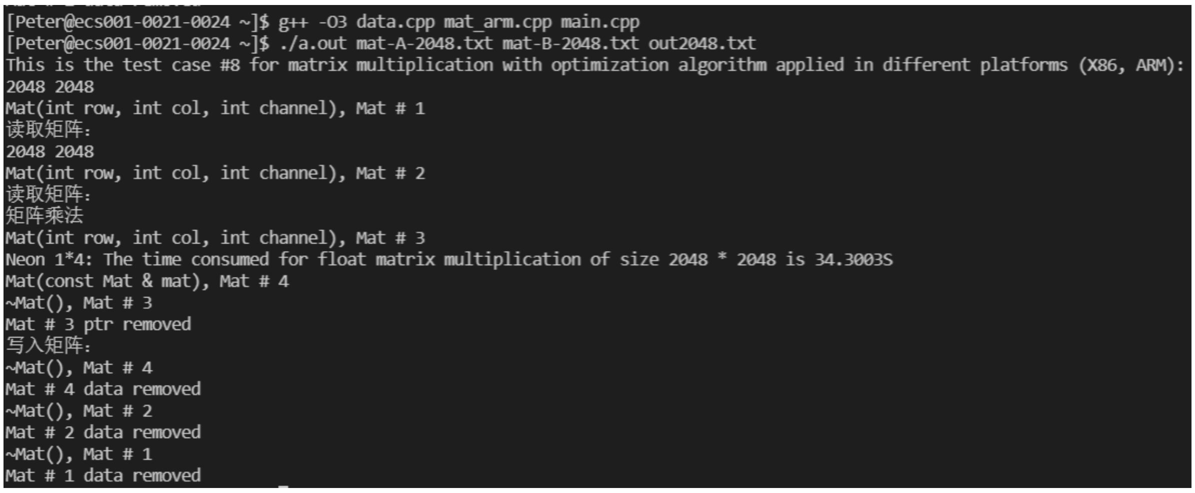
① Compared with the general $i-k-j$ loop $(114.419s)$, $Neon\ 1*4$ is much **faster**. This is opposite on $X86$ platform.

② The **accuracy** of $Neon\ 1*4$ is better since in each loop we compute 4 terms and add them up, which **decreases the rounding error**. The result of the matrix multiplication by using $Neon,\ 1*4$ is shown bellow, you can compare it with that when the general $i-k-j$ loop applied.



```
5252342.00 5053430.00 5087803.00 5006592.00 5185736.50 5085111.50 5122404.00 5107456.50 5100698.00 4986781.50 5179254.00 5227449
070603.50 5118338.00 5046065.50 5139213.50 5186727.50 5093527.50 5170155.50 5121505.00 5160791.50 5198960.50 5003210.00 5152996.5
68037.00 5169701.00 5083414.50 5125027.50 5121347.00 5100276.50 5155154.50 5205806.00 5077904.00 5095979.50 5120667.00 5097434.50
3660.00 5171403.50 5039035.50 4988243.50 5115017.00 5169019.50 5023415.50 5166962.00 5121282.00 5157201.00 4916838.00 5135807.00
580.50 5182358.00 5078802.00 5014951.00 4992185.50 5295681.50 5082729.00 5069696.50 5054841.50 5105380.00 5139349.00 5187010.00 5
34.00 5170546.50 5214462.00 5070264.50 5061777.50 5141278.50 4986247.00 5002703.00 5130706.50 5195514.00 5082265.00 4973122.00 51
6.50 5031055.50 5018507.00 5018176.00 5071361.50 5123651.50 4986940.00 5039760.00 5070902.00 5083931.50 5192630.50 5023740.50 508
.50 5144770.00 5047088.50 5159016.50 5271159.50 4984842.50 5116628.50 5215340.00 5068284.00 5143089.00 5108583.00 5152616.00 5033
00 5155781.50 5058088.00 5038270.50 5163696.50 4963785.50 5052060.50 4930853.00 5163168.00 5188594.50 5022977.00 5044710.00 5157
0 5147873.50 5051299.50 5110213.00 5208332.00 4990139.00 5018981.00 5242519.50 4955542.00 5203041.00 5037024.00 4941249.50 50929
5119933.00 5060302.00 5059407.50 5222085.00 5196563.00 5031897.00 5090158.00 5074506.50 5054061.50 5028606.50 5191555.00 5084697
068618.00 5081065.50 5183447.00 5214607.00 5110343.50 5078855.50 5197078.50 5102453.00 5153607.50 4945785.50 5069129.50 5090439.0
76565.00 5131006.50 5201810.50 5098531.00 5132262.50 5068528.00 5117570.50 5174807.00 5251741.00 5118290.00 5205671.50 5197916.50
4326.50 5109497.50 5042181.50 5176371.00 4968347.50 5141552.00 5083586.50 5047786.00 5137963.50 5063527.50 5242907.50 5115850.00
496.50 5028766.50 5106201.00 5219369.00 5141648.00 5156917.50 5087376.50 5122066.50 5092312.50 5139851.50 4990053.50 5007749.00 5
25.00 5116255.00 5106769.00 5148297.00 5080480.50 5056806.50 5205519.00 5236256.50 5039898.50 5049603.50 5177701.50 5040585.00 51
8.50 5251139.00 5013745.50 5079565.00 5069677.00 5188823.50 5258309.50 5200326.50 5207669.50 5115567.50 5183048.00 5118086.50 521
.00 5076320.00 5152082.00 5173491.00 5077188.00 5173371.50 5277457.00 5032177.00 5251104.00 5034592.50 5056917.50 4980180.00 5128
00 5053932.50 5102772.00 5087265.50 5091222.00 5078676.00 5010356.00 5094428.50 5081229.00 5082207.50 5102952.50 5048732.50 50184
0 5068177.00 5100790.50 5059834.00 5081591.50 5017618.50 5241955.50 5143880.00 5077988.00 5094349.00 5304026.00 5198112.50 52167
5163141.00 5136420.00 5183327.50 5114961.00 5078838.50 4966272.50 5096195.00 5038966.00 5114525.00 5093251.00 5111144.00 5166462
156215.00 5040962.50 5084376.00 5066092.00 5180401.00 5157540.00 5020082.00 5067399.00 5123930.50 5146795.00 5035294.50 5035834.5
5087197.50 4995320.00 5043293.00 4986767.00 5186428.50 5009399.00 5017116.50 4946205.00 5051623.00 4944150.50 5112840.50 5108315
```

③ We can speed up the matrix multiplication by using $-O3$ to compile the code, and the result is shown as follows:



```
[Peter@ecs001-0021-0024 ~]$ g++ -O3 data.cpp mat_arm.cpp main.cpp
[Peter@ecs001-0021-0024 ~]$ ./a.out mat-A-2048.txt mat-B-2048.txt out2048.txt
This is the test case #8 for matrix multiplication with optimization algorithm applied in different platforms (X86, ARM):
2048 2048
Mat(int row, int col, int channel), Mat # 1
读取矩阵:
2048 2048
Mat(int row, int col, int channel), Mat # 2
读取矩阵:
矩阵乘法
Mat(int row, int col, int channel), Mat # 3
Neon 1*4: The time consumed for float matrix multiplication of size 2048 * 2048 is 34.3003S
Mat(const Mat & mat), Mat # 4
~Mat(), Mat # 3
Mat # 3 ptr removed
写入矩阵:
~Mat(), Mat # 4
Mat # 4 data removed
~Mat(), Mat # 2
Mat # 2 data removed
~Mat(), Mat # 1
Mat # 1 data removed
```

We can see that the computation is a lot faster.

◉ **Compare the result of two optimization algorithm on two platforms:**

① The results of $2048 * 2048$ matrix multiplication on $X86$ and ARM platform are the same, which are more accurate than those with the general $i - k - j$ loop matrix multiplication overloaded.

② The only difference when the optimization algorithm applied for these two platforms is the computation speed. After several tests, we conclude that ARM is **faster** than $X86$ in general in this case, which is opposite for the general $i - k - j$ loop case.

We collect the related data for the above test cases in the following table:

Size	2048
time consumption (i-k-j, X86)	95.1331 s
time consumption (i-k-j, ARM)	152.109 s
time consumption (SSE 1*4, X86)	139.729 s
time consumption (SSE 1*4, X86, -O3)	68.1611 s
time consumption (Neon 1*4, ARM)	66.5066 s
time consumption (Neon 1*4, ARM, -O3)	34.3003 s

From this table, we can easily check the analyses and conclusions we have made above.

◉ If we compare these results with those in **previous Project**, we find that the general $i - k - j$ loop for matrix multiplication here is much **slower**. In previous project, the general $i - k - j$ loop for matrix multiplication will just take $35\ s$ approximately, and after using $-O3$ to compile the code, the time cost will decrease to $2\ s$! One possible explanation is that we add the member variable $channel$ in this project, so there are actually 4 loops to compute the matrix multiplication, which is much more time-consuming. Moreover, to realize **soft copy**, we design the class:: $Data$. Then we need use pointers twice to get the matrix data, which is really complex and has high cost.

Part 4 – Difficulties & Solutions

In this part, some difficulties are listed below, most of which are explained or solved in $Part\ 3$. To some extent, this part is like a summary.

a. Some Difficulties

▣ How to realize **soft copy** without causing memory problems?

▣ How to check whether the memory management is successful?

▣ How to realize ROI with **soft copy** applied as the region may not be continuous memory?

▣ How to realize the function of member variable $channel$?

▣ How to fix the result if the data is so large that it causes outflow?

▣ How to realize that this matrix class can work with different data types with $union$ of pointers used?

b. Some Appropriate Solutions

✔ In $Part\ 2, Part\ 3$ and $Test\ case\ \#4$, we explain the idea and test the **soft copy**. The idea is borrowed from **smart pointers: shared_ptr**, which is adding one variable to count the number of pointers pointing to the space of matrix data. Then rewrite the destructor of class:: Mat to release the memory just **once** to avoid **double free**. Also, the copy constructor should be redesigned to avoid **memory leak**.

✔ To check whether the memory management is successful, we just add the output of "delete ptr" or "delete data" to represent that the destructor deletes the pointer or the matrix data, respectively. To identify whose pointer or data is deleted, we add one member variable mat_num. Moreover, we add one **static** variable $count$ and one member variable mat_num to manage the existing matrices in class:: Mat. In this case, matrices that have been deleted will not have a number, so we can identify the remaining matrices easily.

✔ Using the idea of **soft copy**, we should not storage the copied data in a new memory. Instead, we should revise the data in ROI through pointers. When we use $for\ loop$ to move the pointer, its access to memory is continuous. However, the ROI may NOT be continuous space. Here, we still let the pointer move in the $for\ loop$. When the pointer moves into the ROI, we change the data. Otherwise, the data remains still. The drawback is that we must experience unnecessary loops, which is time-consuming.

✔ The basic meaning of $channel$ is that each data of the matrix is actually a **vector** rather than single element. It is equivalent to enlarge the memory of data by corresponding times. Since there are several channels for a matrix element, we may use $channel\ 1$ of matrix A to operate with $channel\ 2$ of matrix B, leading to much more possible results when general operations of matrices applied. To make the matrix operation meaningful, we design the operation of matrix as the same channel number operated, $i.e.$ we only allow $channel\ 1$ of matrix A operates with $channel\ 1$ of matrix B.

✔ It is necessary to consider the outflow case as the basic data type here can just express limited range of data. To avoid the outflow case, we can do **string multiplication** introduced in $Project\ \#1$.

⚑ The **class template** can support different kinds of data, but it needs to define more member variables and thus much more space consumption. In this project, we define a **macro** $myType$ to change the data type by hand, which is not wise but can save a lot space compared with that of **class template**. In $cv::Mat$ in $OpenCV$, it uses **union** containing 5 pointers of different data types pointing to matrix data, which share the same memory. This is a good idea, but when I try to realize it, there are many problems. For example, when we define the getter member functions of class:: $Data$ to get the matrix data, we need to judge the data type ahead. However, the data is given in **main.cpp**. If we do not judge the data type ahead, we must define the getter member functions for all data types, which violates the original idea of **union**, make it lose the original advantages.