实验三 稀疏矩阵的三元组表示

班级：计算机科学与技术(辅修)231 班

姓名：杨雨劼

学号：5601121157

一、前言

在解决数学应用问题的过程中，经常会使用到矩阵(Matrix).若一个矩阵中含有大量的零元素，且非零元素的分布无规律，则称该矩阵为稀疏矩阵(Sparse Matrix).若把稀疏矩阵的所有元素用常规分配方法在计算机中储存，将会产生大量的内存浪费，而且在访问和操作的时候也会造成大量时间开销.通过三元组(Three Tuple)对存储稀疏矩阵内的元素进行压缩存储可以大大节省存储内存和访问操作所需的时间开销.

二、算法设计和实现

为简便操作，规定矩阵都为方阵.

本实验使用单链表的数据结构实现对三元组的定义和操作；同时实现通过三元组对所存储的矩阵进行矩阵的简单操作：矩阵的转置、矩阵的加减法、矩阵的数乘.

使用面向对象的理念进行本次实验的程序设计.

I. 三元组的定义

定义一个三元组结点的类，包含的元素有：行号、列号、存储值和指向下一个结点的指针，相关操作包含结点的创建与销毁；

定义一个三元组链表的类，包含的元素有：头指针、链表的长度；

定义一个方阵类，包含的元素有一个二维数组表示的方阵.

II. 三元组相关操作的定义

1. 三元组链表结点类

(1) 链表结点的构造；

2. 三元组链表类

(1) 初始化链表；

(2) 删除链表；

(3) 转置三元组；

(4) 更新三元组信息；

(5) 算数操作符+/-/=/*的重载；

3. 方阵类

(1) 建立零方阵；

(2) 随机生成非零方阵；

(3) 将方阵信息存储到三元组中；

(4) 通过三元组打印方阵；

以上操作均在头文件 ***mytuple.h*** 中定义，以实现类的封装：

Mytuple.h

```
//
// Created by YangYujie on 2023/11/13.
//
#include <iostream>

#ifndef EXPERIMENT4_MYTUPLE_H
#define EXPERIMENT4_MYTUPLE_H

//方阵大小
#define SIZE 3

using std::cout, std::cin, std::endl;

namespace TUPLE {
    class TupleNode {
        //声明三元组结点类
        int row_;
        int column_;
        double value_;
        TupleNode *next_;
    public:
        TupleNode() {}; //默认构造函数
        TupleNode(int row, int column, double value):
        row_(row), column_(column), value_(value), next_(nullptr) {}
//含参构造函数：列表初始化结点

        friend class Mytuple;   //友元类：Mytuple
        friend class Mymatrix;  //友元类：Mymatrix
    };

    class Mytuple {
        //声明三元组链表类
        TupleNode *header_;
        int length_;
    public:
        Mytuple(): header_(nullptr), length_(0) {}  //列表初始化链表
        ~Mytuple() {
            delete header_;
        }//析构函数
```

```
        static void transpose(Mytuple &A);  //转置三元组
        static void reloadTuple(Mytuple &A);    //更新链表信息和删去链表中零元素结点
        //运算符重载
        Mytuple &operator + (const Mytuple &B);
        Mytuple &operator - (const Mytuple &B);
        Mytuple operator * (double multiplier) const;
        //'*'运算符重载：使用友元函数成为非类成员函数，左操作数可以不为类，返回另一种重载实现double * 三元组
        friend Mytuple operator * (double multiplier, const Mytuple &B) {
            return B * multiplier;
        }
        Mytuple& operator = (const Mytuple &B);

        friend class Mymatrix;  //友元类：Mymatrix
    };

    class Mymatrix {
        //声明方阵类
        double matrix[SIZE][SIZE];
    public:

        Mymatrix(); //构造函数
        ~Mymatrix() {}; //析构函数

        void getRandomMatrix(); //随机生成非零方阵
        void storeInTuple(Mytuple &T);  //将方阵信息存储到三元组中
        void showMatrix (Mytuple &T);   //通过三元组打印方阵
        friend class Mytuple;   //友元类：Mytuple
    };
}
#endif //EXPERIMENT4_MYTUPLE_H
```

III. 相关操作的实现

1. mytuple 类

(1) 转置三元组：

算法思想：创建一个指针变量指向首元结点，对三元组的链表进行遍历操作.三元组中的每个结点中的行、列信息进行互换，即可实现非零元素转换到对应转置方阵位置的操作；零元素在方阵转置前后，总体上看位置不变，故不需要对零元素进行操作.

具体实现如下：

```
void Mytuple::transpose (Mytuple &A)  {
    TupleNode* p = A.header_;
    //遍历链表，交换行列信息
    while (p) {
```

```
        int tempIndex = p->row_;
        p->row_ = p->column_;
        p->column_ = tempIndex;
        p = p->next_;
    }
    cout << "<Mytuple> The Matrix has been transposed."
         << endl << endl;
}
```

(2) 更新三元组信息

两个方阵对应的三元组进行加减操作之后，可能会出现两个非零元素之和或之差为零的情况，需要在完成运算后对链表进行更新，删除含有值为零的结点，以达到节省存储空间的目的.此外，三元组在运算过程中会频繁进行删除节点和添加节点的操作，为了方便，统一在完成操作后进行链表长度的更新.在这个操作中还增加了显示当前方阵更新后删除值为零的结点的个数.

算法思想：遍历从头查找链表中所有值为零的结点，并对其进行删除操作，统计删除结点的个数；完成全部值为零的结点的删除操作后再次从头遍历一遍链表，计算链表当前的长度.

具体实现如下：

```
void Mytuple::reloadTuple(Mytuple &A) {
    TupleNode* p = A.header_, *pPre = A.header_;
    if (!A.header_) {
        //三元组为空，无法删除结点
        cout << "The tuple has no element." << endl;
    }
    else {
        int i = 0, zeroElem = 0;
        while (p) {
            if (p->value_ == 0) {
                //删除零元素
                if (p == A.header_) {
                    //被删除元素是头指针所指的结点
                    p = p->next_;
                    delete A.header_;
                    A.header_ = p;
                }
                else {
                    //被删除元素不是头指针所指的结点
                    pPre->next_ = p->next_;
                    TupleNode* temp = p;
                    p = p->next_;
                    delete temp;
                }
                zeroElem++;
```

```
            }
            else {
                //遍历链表
                p = p->next_;
                if (i >= 1) {
                    pPre = pPre->next_;
                }
                i++;
            }
        }
        if (zeroElem) {
            //ZeroElem 记录被删除的零元素个数
            cout << "<DeleteZeroElement> ";
            cout << zeroElem << " 0-element was deleted from the tuple." << endl;
        }
    }
    p = A.header_; int newLength = 0;
    while (p) {
        //遍历链表，更新链表长度信息
        newLength++;
        p = p->next_;
    }
    A.length_ = newLength;
}
```

(3) 算数操作符+的重载：

算法思想：两个方阵相加即对应元素相加：两个零元素相加仍为零元素；一个零元素和一个非零元素相加的值为非零元素；两个非零元素相加的值为它们的和.因此重载函数的核心是将两个三元组链表进行合并.

注意到函数声明：形参为常引用，因此不能修改处在右操作数的三元组链表；该成员函数隐式调用对象(即处在左操作数的三元组对象)，函数返回类型为非 const 的类引用类型，是因为函数通过使用*this 使返回的对象为被调用对象的拷贝而非被调用对象本身，因此该拷贝可以被修改，而被调用对象本身不可被修改.

具体实现如下：

```
//'+'运算符重载：左操作数可变，右操作数不可变(const  &)
Mytuple& Mytuple::operator + (const Mytuple &B) {
    //算法思想：两个无序单链表的合并，原被加数链表指针指向新链表，加数链表不可改动
    TupleNode *pa = nullptr, *paPre = nullptr, *pb = B.header_;
    while (pb) {
        pa = this->header_, paPre = header_;
        int i = 0;
        //加数指针不变，遍历一遍被加数链表
```

```
        while (pa) {
            //行列信息相同，对应值相加
            if (pa->row_ == pb->row_ && pa->column_ == pb->column_) {
                pa->value_ += pb->value_;
                break;
            }
            pa = pa->next_;
            if (i >= 1) //指向指针的前一个结点
                paPre = paPre->next_;
            if (!pa) {
                //遍历完整个链表都没有相同元素，则在被加数表尾添加结点实现0+加数
                TupleNode *newNode = new TupleNode(pb->row_, pb->column_, pb->value_);
                paPre->next_ = newNode;
            }
            i++;
        }
        pb = pb->next_;
    }
    return *this;
}
```

(4) 算数操作符-的重载：

算法思想：与加号操作符的重载类似.

具体实现省略.

(5) 算数操作符*的重载：

通过重载乘号操作符实现浮点数与三元组的相乘，即实现矩阵的数乘运算.

算法思想：从头遍历三元组链表，对每个结点的值乘上浮点数.若乘的数为零，在更新操作时可以对链表进行再次修改.

第一种重载是：三元组 * double，此时函数形参是右操作数 double 类型，而左操作数的类会由函数隐式调用，使用局部变量作为函数的返回值可以保证原对象不被修改的同时，如果下一个操作是赋值可以直接将返回值赋给对应的对象.

```
/* '*'运算符重载：第一种情况：三元组 * double
 * 隐式调用对象(使用this)，返回对象的拷贝且不能改变对象，  函数体用const 修饰
 */
Mytuple Mytuple::operator * (double multiplier) const
{
    //算法思想：遍历一遍链表，对所有元素都乘上乘数
    Mytuple temp;
    TupleNode* p = this->header_;
    while (p) {
```

```
        p->value_ *= multiplier;
        p = p->next_;
    }
    temp = *this;
    return temp;
}
```

在 Mytuple 类内用友元函数定义另一种乘号操作符的重载，该友元函数不再是类成员函数而是非类成员函数，返回值可以直接用重载完成的三元组*double.

具体实现如下：

```
//'*'运算符重载：使用友元函数成为非类成员函数，左操作数可以不为类，返回另一种重载实现double  * 三元组
friend Mytuple operator * (double multiplier, const Mytuple &B) {
    return B * multiplier;
}
```

(6) 赋值运算符的重载：

算法思想：赋值前后右操作数会改变，比起在原链表的基础上进行增加删除结点操作实现链表的赋值，将原链表进行销毁后重新创建并增加结点会更便捷.

当左右操作数相同时，即自我赋值，应该直接返回；在进行操作前先检查原链表是否为空，非空则进行销毁；此后可以从头建立链表并依次添加右操作数链表的所有结点，从而实现三元组赋值操作.

具体实现如下：

```
//'='运算符重载：将右操作数的对象拷贝给左操作数的对象
Mytuple& Mytuple::operator = (const Mytuple &B)
{
    //左右操作数相同：自我赋值
    if (this == &B) {
        return *this;
    }

    //防止内存泄露
    if (this != &B) {
        if (header_ != nullptr) {
            delete header_;
            header_ = nullptr;
        }
    }

    TupleNode* p = B.header_;
    //重新对调用对象建立链表，使元素和右操作数的对象的元素一样
    while (p) {
        TupleNode* newNode = new TupleNode(p->row_, p->column_, p->value_);
```

```
        if (!this->header_) {
            this->header_ = newNode;
        } else {
            newNode->next_ = this->header_;
            this->header_ = newNode;
        }
        p = p->next_;
    }
    return *this;
}
```

2. Mymatrix 类

该类的成员函数都是以遍历链表或方阵的方法实现的，所以具体实现不再展示.

(1) 随机生成非零方阵

算法思想：对零方阵的每一个元素进行随机数赋值；如果遍历赋值后仍为零方阵，则重新进行一轮赋值.

(2) 将方阵信息存储到空三元组中

算法思想：对方阵进行遍历，如果遇到非零元素，则在三元组链表中新建一个结点存储相关信息.

在传入三元组形参后，对其建立链表之前，首先检验传入的三元组是否为空，若非空则删除原链表.

(3) 打印方阵

算法思想：在打印方阵之前先更新一次方阵的信息；创建一个新的零方阵，对链表进行遍历并对每个结点所对应的方阵元素进行赋值，遍历完链表后该方阵即为三元组对应的方阵；遍历一遍方阵打印出方阵的元素，并根据方阵信息输出非零元的个数.

主程序中设计了生成方阵、存储方阵、转置方阵和对方阵进行运算的流程：

main.cpp

```
#include <iostream>
#include <ctime>
#include "mytuple.h"
using namespace TUPLE;
using namespace std;

int main() {
    cout << fixed;
    cout.precision(2);
    time_t t;
    srand((unsigned int) time(&t));

    cout << "<Create Two Matrix>" << endl;
```

```
    string matrixA, matrixB;
    cout << "Name the matrixA: ";   cin >> matrixA;
    cout << endl;
    cout << "Name the matrixB: ";   cin >> matrixB;
    cout << endl;
    Mymatrix A;
    Mytuple tupleA;
    Mymatrix B;
    Mytuple tupleB;

    cout << '<' << matrixA << ">" << endl;
    A.getRandomMatrix();
    A.storeInTuple(tupleA);

    cout << '<' << matrixB << ">" << endl;
    B.getRandomMatrix();
    B.storeInTuple(tupleB);

    tupleB = 3 * tupleB - tupleA;

    cout << '<' << matrixA << ">" << endl;
    Mytuple::transpose(tupleA);
    A.showMatrix(tupleA);
    cout << '<' << matrixB << ">" << endl;
    B.showMatrix(tupleB);

    return 0;
}
```

三、程序运行结果

```
<Create Two Matrix>
Name the matrixA: First

Name the matrixB: Second

<First>
<Mymatrix> The matrix has been initialized:
0.00  9.00  7.00
0.00  4.00  4.00
0.00  0.00  0.00

<Mymatrix> The [matrix] has been stored into [tuple].

<Second>
<Mymatrix> The matrix has been initialized:
0.00  0.00  0.00
0.00  0.00  4.00
0.00  7.00  0.00

<Mymatrix> The [matrix] has been stored into [tuple].
```

图 1 创建两个随机方阵并存储到三元组中

程序开始时，用户可以对两个方阵进行命名.如图 1 所示，将这两个方阵命名为<First>和<Second>.头文件中宏定义方阵大小为 5，使用随机数生成了这两个大小为 3 的非零方阵：

$$First = \begin{matrix} 0.00 & 9.00 & 7.00 \\ 0.00 & 4.00 & 4.00 \\ 0.00 & 0.00 & 0.00 \end{matrix}$$

$$Second = \begin{matrix} 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 4.00 \\ 0.00 & 7.00 & 0.00 \end{matrix}$$

根据程序流程可知，求 First 矩阵的转置结果为：

$$First^T = \begin{matrix} 0.00 & 0.00 & 0.00 \\ 9.00 & 4.00 & 0.00 \\ 7.00 & 4.00 & 0.00 \end{matrix}$$

而运算表达式 Second= 3 * Second – First(这里的 First 未被转置)的结果为：

$$Second = \begin{matrix} 0.00 & -9.00 & -7.00 \\ 0.00 & -4.00 & 8.00 \\ 0.00 & 21.00 & 0.00 \end{matrix}$$

```
<First>
<Mytuple> The Matrix has been transposed.

<Mytuple> Show Matrix
0.00    0.00    0.00
9.00    4.00    0.00
7.00    4.00    0.00
Numbers of Non-0 Element: 4

<Second>
<Mytuple> Show Matrix
0.00    -9.00    -7.00
0.00    -4.00    8.00
0.00    21.00    0.00
Numbers of Non-0 Element: 5
```

图 2 进行 Second=3*Second-First;和 First 转置的运算结果

在程序流程中，先进行了 Second=3*Second-First;再进行 First 转置.上方的结果是转置后的 First 矩阵，与预期一致；下方的结果是运算结果，与预期一致.同时还输出了各个矩阵非零元的个数.

四、实验分析与总结

在编写程序的时候，遇到了许多问题：

1. 首次使用面向对象的风格编写程序，对类的相关知识掌握不是很熟悉.在设计结点类、链表类和矩阵类时，思考是否应该把它们的关系定为并列关系；如果能更好的话，是否要用到类的继承相关的知识.这样在需要调用其他类的成员函数时，可以通过继承的关系改变成员函数的访问权限.而不是简简单单地设计三个友元类，将原本方法的私有权限给破坏掉了；

2. 在设计运算符重载时，对形参和函数返回类型的掌握不太熟悉.而这可能和 const 关键字、引用、this 指针等知识的掌握程度有关.所以在设计运算符重载时，要多去总结各种运算符重载的设计规律.尤其是第一次设计赋值运算符重载时，完全没有考虑到自我赋值和被赋值对象非空的情况，在进行这种和内存管理息息相关的操作时应时刻保持警惕；

3. 在其他运用到链表的操作时，第一次设计选择的是通过使用带头结点的链表的数据结构来存储三元组信息，有些算法完全可以使用不带头结点的链表进行实现；

4. 在算法设计方面要考虑周到.有些时候设计出的算法并不完备，比如第一次设计两个方阵相加时，误以为找到两个链表重复元素进行相加即可，但实际上忽略了 0+非零的情况，而且在设计时误以为链表是以行优先的序号递增顺序存储结点，但实际上链表的存储顺序是无序的；有些时候设计出的算法过于复杂，比如第一次设计赋值运算时，想法是在被赋值对象的链表上进行替换节点并将多余结点删除(缺少的结点则添加)，但实际上忽略了被赋值的对象应该为空，只需在空表上一个一个复制

结点即可.

总之，尝试使用面向对象的风格是我的突破，但仍需要大量练习才能把数据结构和面向对象这两方面掌握到位.