```
//============================================================
// FILE: vector3d_t.h
// Name: David Tu
// Program Description: 3D Vector Class Implementation. This class is able to construct 3D Vectors
// as well as be able to perfrom mathematical computations on 3D Vectors
//      Input: 3D Vectors, scalars
//      Output: 3D Vectors (After computation)
//============================================================

#pragma once
#ifndef __vector3d_T_H__
#define __vector3d_T_H__

#include <iostream>
#include <string>
#include <cmath>
#include <initializer_list>

template <typename T> class vector3d;
template <typename T> std::ostream&  operator<<(std::ostream& os, const vector3d<T>& v);

typedef vector3d<double> vector3dD;
typedef vector3d<float> vector3dF;
typedef vector3d<int> vector3dI;
typedef vector3d<long> vector3dL;

template <typename T>
class vector3d{
public:
        vector3d();
        vector3d(const std::string& name, int dims);
        vector3d(const std::string& name, int dims, const std::initializer_list<T>& li);
//------------------------------------------------------------------------
        T operator[](int i) const;
        T& operator[](int i);
//------------------------------------------------------------------------
        void name(const std::string& name);
        const std::string& name() const;
//------------------------------------------------------------------------
        vector3d<T>& operator+=(const vector3d<T>& v);
        vector3d<T>& operator-=(const vector3d<T>& v);
//------------------------------------------------------------------------
        vector3d<T>& operator+=(T k);
        vector3d<T>& operator-=(T k);
        vector3d<T>& operator*=(T k);
        vector3d<T>& operator/=(T k);
//------------------------------------------------------------------------
        vector3d<T> operator-();
        vector3d<T> operator+(const vector3d<T>& v);
        vector3d<T> operator-(const vector3d<T>& v);
//------------------------------------------------------------------------
        friend vector3d operator+(T k, const vector3d& v){
                return vector3d(std::to_string(k) + "+" + v.name_, v.dims_, {k + v[0], k + v[1], k + v[2], 0});
        }
        friend vector3d operator+(const vector3d& v, T k){
                return  k + v;
        }
        friend vector3d operator-(const vector3d& v, T k){
                return -k + v;
        }
        friend vector3d operator-(T k, const vector3d& v){
                return vector3d(std::to_string(k) + "-" + v.name_, v.dims_, {k - v[0], k - v[1], k - v[2], 0});
        }
        friend vector3d operator*(T k, const vector3d& v){
                return vector3d(std::to_string(k) + "*" + v.name_, v.dims_, {k * v[0], k * v[1], k * v[2], 0});
        }
        friend vector3d operator*(const vector3d& v, T k){
                return k * v;
        }
        friend vector3d operator/(const vector3d& v, T k){
                return vector3d(v.name_ + "/" + std::to_string(k), v.dims_, {v[0]/k, v[1]/k, v[2]/k, 0});
        }
```

```
//-------------------------------------------------------------------------
        bool operator==(const vector3d<T>& v) const;
        bool operator!=(const vector3d<T>& v) const;
//-------------------------------------------------------------------------
        T dot(const vector3d<T>& v) const;
        T magnitude() const;
        T angle(const vector3d<T>& v) const;
        vector3d<T> cross(const vector3d<T>& v) const;
//-------------------------------------------------------------------------
        static vector3d<T> zero();
//-------------------------------------------------------------------------
        friend std::ostream& operator<< <>(std::ostream& os, const vector3d<T>& v);
private:
        void check_equal_dims(const vector3d<T>& v) const;
        void check_bounds(int i) const;
private:
        constexpr static double EPSILON = 1.0e-10;
        std::string name_;
        int dims_;
        T data_[4];
};

//-------------------------------------------------------------------------
template <typename T> vector3d<T>::vector3d() : vector3d("", 3){// 3d default dims}
template <typename T> vector3d<T>::vector3d(const std::string& name, int dims): name_(name), dims_(dims){
        std::memset(data_, 0, dims_ * sizeof(T));
        data_[3] = T();  // vectors have 0 at end, pts have 1
}
template <typename T> vector3d<T>::vector3d(const std::string& name, int dims, const std::initializer_list<T>& li):
vector3d(name, dims){
        int i = 0;
        for (T value : li){
                if (i > dims_){
                        break;
                }
                data_[i++] = value;
        }
        data_[3] = T();
}
//-------------------------------------------------------------------------
template <typename T> T vector3d<T>::operator[](int i) const{// read-only index operator
        check_bounds(i);
        return data_[i];
}
template <typename T> T& vector3d<T>::operator[](int i){// read-write index operator
        check_bounds(i);
        return data_[i];
}
//-------------------------------------------------------------------------
template <typename T> void vector3d<T>::name(const std::string& name){
        name_ = name;
}
template <typename T> const std::string& vector3d<T>::name() const{
        return name_;
}
//-------------------------------------------------------------------------
template <typename T> vector3d<T>& vector3d<T>::operator+=(const vector3d<T>& v){
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i){
                u[i] += v[i];
        }
        return *this;
}
template <typename T> vector3d<T>& vector3d<T>::operator-=(const vector3d<T>& v){
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i) {
                u[i] -= v[i];
        }
        return *this;
}
//-------------------------------------------------------------------------
template <typename T> vector3d<T>& vector3d<T>::operator+=(T k){
```

```
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i) {
                u[i] += k;
        }
        return *this;
}
template <typename T> vector3d<T>& vector3d<T>::operator*=(T k){
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i) {
                u[i] *= k;
        }
        return *this;
}
template <typename T> vector3d<T>& vector3d<T>::operator-=(T k){
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i) {
                u[i] -= k;
        }
        return *this;
}
template <typename T> vector3d<T>& vector3d<T>::operator/=(T k){
        vector3d<T>& u = *this;
        for (int i = 0; i < 3; ++i) {
                u[i] /= k;
        }
        return *this;
}
//-----------------------------------------------------------------------
template <typename T> vector3d<T> vector3d<T>::operator-(){
        return vector3d<T>("-" + name_, dims_, {-data_[0], -data_[1], -data_[2], 0});
}
template <typename T> vector3d<T> vector3d<T>::operator+(const vector3d& v){
        const vector3d<T>& u = *this;
        check_equal_dims(v);
        return vector3d<T>(u.name_ + "+" + v.name_, dims_, {u[0] + v[0], u[1] + v[1], u[2] + v[2], 0});
}
template <typename T> vector3d<T> vector3d<T>::operator-(const vector3d<T>& v){
        check_equal_dims(v);
        return vector3d<T>(name_ + "-" + v.name_, dims_, {data_[0] - v[0], data_[1] -v[1], data_[2] - v[2], 0});
}
//-----------------------------------------------------------------------
template <typename T> bool vector3d<T>::operator==(const vector3d<T>& v) const{
        const vector3d<T>& u = *this;
        check_equal_dims(v);
        return  std::abs(u[0] - v[0]) < vector3d<T>::EPSILON && std::abs(u[1] - v[1]) <
        vector3d<T>::EPSILON && std::abs(u[2] - v[2]) < vector3d<T>::EPSILON;
}
template <typename T> bool vector3d<T>::operator!=(const vector3d<T>& v) const{
        return !(*this == v);
}
//-----------------------------------------------------------------------
template <typename T> T vector3d<T>::dot(const vector3d<T>& v) const{
        check_equal_dims(v);
        T dot = 0;
        for (int i = 0; i < dims_; ++i) {
                dot += data_[i] * v.data_[i];
        }
        return dot;
}
template <typename T> T vector3d<T>::magnitude() const{
        return sqrt(dot(*this));
}
template <typename T> T vector3d<T>::angle(const vector3d<T>& v) const{
        T dot = this->dot(v);
        T mag = this->magnitude();
        T othermag = v.magnitude();
        return acos(dot / (mag * othermag));
}
template <typename T> vector3d<T> vector3d<T>::cross(const vector3d<T>& v) const{
        const vector3d<T>& u = *this;
        check_equal_dims(v);
        if (v.dims_ != 3){
```

```
                throw new std::invalid_argument("cross_product only implemented for vector3d's");
        }
        return vector3d(name_ + " x " + v.name_, dims_,
        {u[1]*v[2] - u[2]*v[1], -(u[0]*v[2] - u[2]*v[0]), u[0]*v[1] - u[1]*v[0], 0 });
}
//---------------------------------------------------------------------
template <typename T> vector3d<T> vector3d<T>::zero(){
        return vector3d("zero", 3, { 0, 0, 0, 0 });
}
//---------------------------------------------------------------------
template <typename T> std::ostream& operator<<(std::ostream& os, const vector3d<T>& v){
        os << "<'" << v.name_ << "', ";
        if (v.dims_ == 0){
                os << "empty>";
        } else{
                for (int i = 0; i < v.dims_ + 1; ++i){
                        os << v[i];
                        if (i < v.dims_){
                                os << " ";
                        }
                }
                os << ">";
        }
        return os;
}
//---------------------------------------------------------------------
template <typename T> void vector3d<T>::check_equal_dims(const vector3d<T>& v) const{
        if (dims_ != v.dims_){
                throw new std::invalid_argument("vector3d dims mismatch");
        }
}
template <typename T> void vector3d<T>::check_bounds(int i) const{
        if (i > dims_) { // 1 extra dimension for pts/vectors
                throw new std::invalid_argument("out of bounds");
        }
}

#endif

//=============================================================
// end of file: vector3d_t.h
//=============================================================
```

```
//============================================================
// FILE: matrix3d_t.h
// Name: David Tu
// Program Description: 3D Matrix Implementation. This class is able to construct 3D matricies
// as well be able to perform mathematical computations on 3D matricies
//      Input: 3D matricies, 3D vectors, scalars
//      Output: 3D matricies (After computation)
//============================================================

#pragma once

#ifndef __matrix3d_T_H__
#define __matrix3d_T_H__
#include <cstring>
#include "vector3d_t.h"

template <typename T> class matrix3d;
template <typename T> std::ostream& operator<<(std::ostream& os, const matrix3d<T>& m);

typedef matrix3d<double> matrix3dD;
typedef matrix3d<float> matrix3dF;
typedef matrix3d<int> matrix3dI;
typedef matrix3d<long> matrix3dL;

template <typename T>
class matrix3d{
public:
        matrix3d();
        matrix3d(const std::string& name, int dims);
        matrix3d(const std::string& name, int dims, const std::initializer_list<vector3d<T>>& li);
        matrix3d(const std::string& name, int dims, const std::initializer_list<T>& li);
//=========================================================================
        matrix3d<T>& operator=(T array[9]);
        matrix3d<T>& operator=(T k);
//=========================================================================
// indexing ops...
        vector3d<T> operator[](int i) const;
        vector3d<T>& operator[](int i);
        T  operator()(int row, int col) const;
        T& operator()(int row, int col);
        T* opengl_memory();
//=========================================================================
        void name(const std::string& name);
        const std::string& name() const;
//============================  LINEAR ALGEBRA  ===========================
        matrix3d<T>& operator+=(T k);
        matrix3d<T>& operator-=(T k);
        matrix3d<T>& operator*=(T k);
        matrix3d<T>& operator/=(T k);
//=========================================================================
        matrix3d<T>& operator+=(const matrix3d<T>& b);
        matrix3d<T>& operator-=(const matrix3d<T>& b);
//=========================================================================
        matrix3d<T> operator-();
        matrix3d<T> operator+(const matrix3d<T>& b);
        matrix3d<T> operator-(const matrix3d<T>& b);
//=========================================================================
        friend matrix3d operator+(const matrix3d& a, T k){
                return matrix3d(std::to_string(k) + "+" + a.name(), 3, { a[0] + k, a[1] + k, a[2] + k });
        }
        friend matrix3d operator+(T k, const matrix3d& a){
                return  a + k;
        }
        friend matrix3d operator-(const matrix3d& a, T k){
                return  a + -k;
        }
        friend matrix3d operator-(T k, const matrix3d& a){
                return matrix3d(std::to_string(k) + "-" + a.name(), 3, { k - a[0], k - a[1], k - a[2] });
        }
        friend matrix3d operator*(const matrix3d& a, T k){
                return matrix3d(a.name() + "*" + std::to_string(k), 3, { a[0] * k, a[1] * k, a[2] * k });
        }
```

```
        friend matrix3d<T> operator*(T k, const matrix3d& a){
                return a * k;
        }
        friend matrix3d operator/(const matrix3d& a, T k){
                return matrix3d(a.name() + "/" + std::to_string(k), 3, { a[0] / k, a[1] / k, a[2] / k });
        }
//=========================================================================
        friend matrix3d operator*(const matrix3d& m, const vector3d<T>& v){
                matrix3d<T> product(m.name() + "*" + v.name(), 3);
                for (int i = 0; i < 3; ++i) {
                        for (int j = 0; j < 3; ++j) {
                                product.cols_[0][i] += m.cols_[i][j] * v[j];
                        }
                }
                return product;
        }
        friend matrix3d operator*(const vector3d<T>& v, const matrix3d& m){
                return m * v;
        }
        matrix3d<T> operator*(const matrix3d<T>& b);
//=========================================================================
        matrix3d<T> transpose() const;// create a new matrix transpose()
        T determinant() const;
        T trace() const;
//=========================================================================
        matrix3d<T> minors() const;// see defn
        matrix3d<T> cofactor() const;// (-1)^(i+j)*minors()(i, j)
        matrix3d<T> adjugate() const;// cofactor.transpose()
        matrix3d<T> inverse() const;// adjugate()/determinant()
//=========================================================================
        static matrix3d<T> identity(int dims);// identity matrix
        static matrix3d<T> zero(int dims);// zero matrix
//=========================================================================
        bool operator==(const matrix3d<T>& b) const;
        bool operator!=(const matrix3d<T>& b) const;
//=========================================================================
        friend std::ostream& operator<< <> (std::ostream& os, const matrix3d<T>& m);
private:
        void check_equal_dims(const matrix3d<T>& v) const;
        void check_bounds(int i) const;
        void swap(T& x, T& y);
private:
        std::string name_;
        int dims_;
        vector3d<T> cols_[4];
        T data_[16];
};

//=============================================================================================
template <typename T> matrix3d<T>::matrix3d() : matrix3d("", 3){// 3d default dims}
template <typename T> matrix3d<T>::matrix3d(const std::string& name, int dims): name_(name), dims_(dims) {
        for (int i = 0; i < 4; ++i) {
                cols_[i].name("col" + std::to_string(i));
        }
        std::memset(data_, 0, 16 * sizeof(T));
}
template <typename T> matrix3d<T>::matrix3d(const std::string& name, int dims,
                                        const std::initializer_list<vector3d<T>>& li): matrix3d(name, dims) {
        int i = 0;
        for (vector3d<T> value : li) {
                if (i > dims_)  {
                        break;
                }
                cols_[i++] = value;
        }
}
template <typename T> matrix3d<T>::matrix3d(const std::string& name, int dims, const std::initializer_list<T>& li):
matrix3d(name, dims) {
        int i = 0;
        for (T value : li) {
                cols_[i/3][i % 3] = value; ++i;
        }
```

```
}
//=============================================================================================

template <typename T> matrix3d<T>& matrix3d<T>::operator=(T array[9]) {

	for (int i = 0; i < 3; ++i) {
		for (int j = 0; j < 3; ++i) {
			cols_[i][j] = array[i + j];
		}
	}
	return *this;
}
template <typename T> matrix3d<T>& matrix3d<T>::operator=(T k) {
	for (int i = 0; i < 3; ++i) {
		for (int j = 0; j < 3; ++j) {
			cols_[i][j] = k;
		}
	}
	return *this;
}
//=============================================================================================

template <typename T> vector3d<T> matrix3d<T>::operator[](int i) const {
	check_bounds(i);
	return cols_[i];
}
template <typename T> vector3d<T>& matrix3d<T>::operator[](int i) {
	check_bounds(i);
	return cols_[i];
}
template <typename T> T matrix3d<T>::operator()(int row, int col) const {
	check_bounds(row);
	check_bounds(col);
	return cols_[col][row];
}
template <typename T> T& matrix3d<T>::operator()(int row, int col) {
	check_bounds(row);
	check_bounds(col);
	return cols_[col][row];
}
template <typename T> T* matrix3d<T>::opengl_memory() {// constant ptr
	// implement code here
	check_bounds(row);
	check_bounds(col);
	return *cols_[col][row];
}
//=============================================================================================
template <typename T> void matrix3d<T>::name(const std::string& name) {
	name_ = name;
}
template <typename T> const std::string& matrix3d<T>::name() const {
	return name_;
}
//================================  LINEAR ALGEBRA  ================================

template <typename T> matrix3d<T>& matrix3d<T>::operator+=(T k) {
	matrix3d<T>& a = *this;
	name_ = std::to_string(k) + "+" + name_;
	for (int i = 0; i < 4; ++i) {
		a[i] += k;
	}
	return *this;
}
template <typename T> matrix3d<T>& matrix3d<T>::operator-=(T k) {
	*this += -k;
	return *this;
}
template <typename T> matrix3d<T>& matrix3d<T>::operator*=(T k) {
	matrix3d<T>& a = *this;
	name_ = name_ + "*" + std::to_string(k);
	for (int i = 0; i < 4; ++i) {
		a[i] *= k;
```

```
        }
        return *this;
}
template <typename T> matrix3d<T>& matrix3d<T>::operator/=(T k) {
        matrix3d<T>& a = *this;
        name_ = name_ + "/" + std::to_string(k);
        for (int i = 0; i < 4; ++i) {
                a[i] /= k;
        }
        return *this;
}
//=============================================================================================

template <typename T> matrix3d<T>& matrix3d<T>::operator+=(const matrix3d<T>& b) {
        for (int i = 0; i < dims_; ++i) {
                cols_[i] = cols_[i] + b.cols_[i];
        }
        return *this;
}
template <typename T> matrix3d<T>& matrix3d<T>::operator-=(const matrix3d<T>& b) {
        for (int i = 0; i < dims_; ++i) {
                cols_[i] = cols_[i] - b.cols_[i];
        }
        return *this;
}
//=============================================================================================
template <typename T> matrix3d<T> matrix3d<T>::operator-() {
        const matrix3d<T>& a = *this;
        return matrix3d<T>("-" + name_, 3, { -a[0], -a[1], -a[2] });
}
template <typename T> matrix3d<T> matrix3d<T>::operator+(const matrix3d<T>& b) {
        const matrix3d<T>& a = *this;
        check_equal_dims(b);
        return matrix3d<T>(name_ + "+" + b.name_, dims_, { a[0] + b[0], a[1] + b[1], a[2] + b[2] });
}
template <typename T> matrix3d<T> matrix3d<T>::operator-(const matrix3d<T>& b) {
        const matrix3d<T>& a = *this;
        check_equal_dims(b);
        return matrix3d<T>(name_ + "-" + b.name_, dims_, { a[0] - b[0], a[1] - b[1], a[2] - b[2] });
}
//=============================================================================================

template <typename T> matrix3d<T> matrix3d<T>::operator*(const matrix3d<T>& b) {
        const matrix3d<T>& a = *this;
        return matrix3d<T>(a.name_ + "*" + b.name_, 3, {
                a(0,0)*b(0,0) + a(0,1)*b(1,0) + a(0,2)*b(2,0),
                a(1,0)*b(0,0) + a(1,1)*b(1,0) + a(1,2)*b(2,0),
                a(2,0)*b(0,0) + a(2,1)*b(1,0) + a(2,2)*b(2,0),
                a(0,0)*b(0,1) + a(0,1)*b(1,1) + a(0,2)*b(2,1),
                a(1,0)*b(0,1) + a(1,1)*b(1,1) + a(1,2)*b(2,1),
                a(2,0)*b(0,1) + a(2,1)*b(1,1) + a(2,2)*b(2,1),
                a(0,0)*b(0,2) + a(0,1)*b(1,2) + a(0,2)*b(2,2),
                a(1,0)*b(0,2) + a(1,1)*b(1,2) + a(1,2)*b(2,2),
                a(2,0)*b(0,2) + a(2,1)*b(1,2) + a(2,2)*b(2,2) });
}
//=============================================================================================
template <typename T> matrix3d<T> matrix3d<T>::transpose() const {
        const matrix3d<T>& m = *this;
        return matrix3d<T>("Transpose(" + name_ + ")", 3, {
                m(0,0), m(1,0), m(2,0),
                m(0,1), m(1,1), m(2,1),
                m(0,2), m(1,2), m(2,2) });
}
template <typename T> T matrix3d<T>::determinant() const {
        const matrix3d<T>& m = *this;
        return (m(0,0) * ((m(1,1) * m(2,2)) - (m(1,2) * m(2,1))))
                - (m(0,1) * ((m(1,0) * m(2,2)) - (m(1,2) * m(2,0))))
                + (m(0,2) * ((m(1,0) * m(2,1)) - (m(1,1) * m(2,0))));
}
template <typename T> T matrix3d<T>::trace() const {
        const matrix3d<T>& m = *this;
        return m(0,0) + m(1,1) + m(2,2);
```

```
}
//=============================================================================================

// | | e f |  | d f |  | d e | |      Matrix of minors
// | | h i |  | g i |  | g h | |
// |                           |
// | | b c |  | a c |  | a b | |
// | | h i |  | g i |  | g h | |
// |                           |
// | | b c |  | a c |  | a b | |
// | | e f |  | d f |  | d e | |
//-------------------------------------------------------------
template <typename T> matrix3d<T> matrix3d<T>::minors() const {
        const matrix3d<T>& m = *this;
        return matrix3d<T>("Min(" + name_ + ")", 3, {
                (m(1,1)*m(2,2) - m(1,2)*m(2,1)),
                (m(0,1)*m(2,2) - m(0,2)*m(2,1)),
                (m(0,1)*m(1,2) - m(0,2)*m(1,1)),
                (m(1,0)*m(2,2) - m(1,2)*m(2,0)),
                (m(0,0)*m(2,2) - m(0,2)*m(2,0)),
                (m(0,0)*m(1,2) - m(0,2)*m(1,0)),
                (m(1,0)*m(2,1) - m(1,1)*m(2,0)),
                (m(0,0)*m(2,1) - m(0,1)*m(2,0)),
                (m(0,0)*m(1,1) - m(0,1)*m(1,0)) });
}
template <typename T> matrix3d<T> matrix3d<T>::cofactor() const {
        const matrix3d<T>& m = *this;
        return matrix3d<T>("Cofactor(" + name_ + ")", 3, {
                m.minors()(0,0), -m.minors()(0,1), m.minors()(0,2),
                -m.minors()(1,0), m.minors()(1,1), -m.minors()(1,2),
                m.minors()(2,0), -m.minors()(2,1), m.minors()(2,2) });
}
template <typename T> matrix3d<T> matrix3d<T>::adjugate() const {
        return cofactor().transpose();
}
template <typename T> matrix3d<T> matrix3d<T>::inverse() const {
        if (determinant() == 0) {
                throw new std::invalid_argument("Cannot invert because the determinant is zero");
        }
        return adjugate() / determinant();
}
//=============================================================================================

template <typename T> matrix3d<T> matrix3d<T>::identity(int dims) {
        return matrix3d<T>("Identity", 3, { 1, 0, 0, 0, 1, 0, 0, 0, 1 });
}
template <typename T> matrix3d<T> matrix3d<T>::zero(int dims){
        return matrix3d<T>("Zero", dims, { 0, 0, 0, 0, 0, 0, 0, 0, 0 });
}
template <typename T> bool matrix3d<T>::operator==(const matrix3d<T>& b) const {
        check_equal_dims(b);
        const matrix3d<T>& a = *this;
        return a[0] == b[0] && a[1] == b[1] && a[2] == b[2];
}
template <typename T> bool matrix3d<T>::operator!=(const matrix3d<T>& b) const {
        return !(*this == b);
}
//=============================================================================================
template <typename T> std::ostream& operator<<(std::ostream& os, const matrix3d<T>& m) {
        os << "<'" << m.name_ << "', ";
        for (int i = 0; i < 3; ++i) {
                os << m.cols_[i];
        }
        os << "> OR by rows...\n";
        for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                        os << m(i, j) << " ";
                }
                os << "\n";
        }
        return os << ">";
}
```

```
//=============================================================================================

template <typename T> void matrix3d<T>::check_equal_dims(const matrix3d<T>& v) const {
        if (dims_ != v.dims_) {
                throw new std::invalid_argument("matrix3d dims mismatch");
        }
}
template <typename T> void matrix3d<T>::check_bounds(int i) const {
        if (i > dims_) {
                throw new std::invalid_argument("out of bounds");
        }
}
template <typename T> void matrix3d<T>::swap(T& x, T& y) {
        T temp = x;
        x = y;
        y = temp;
}

#endif

//============================================================
// end of file: matrix3d_t.h
//============================================================
```

```
//============================================================
// FILE: main.cpp
// Name: David Tu
// Program Description: The main program. This program tests the 3D Matrix and 3D Vector Classes by using assertions
//      Input: None
//      Output: None
//============================================================

#define _USE_MATH_DEFINES
#include <iostream>
#include <cstring>
#include <initializer_list>
#include <cassert>
#include "matrix3d_t.h"
#include "vector3d_t.h"

template <typename T> void print(T v){std::cout << v << std::endl;}
template <typename T> void show_vect(T v) {std::cout << v.name() << " is: " << v << std::endl;}
template <typename T> void show_mat(T m) {std::cout << m.name() << " is: " << m << std::endl;}

void test_vectors() {
        print("\n==================== TESTING VECTORS ========================");
        vector3dD u("u", 3, { 1, 2, 4 });
        vector3dD v("v", 3, { 8, 16, 32 });
        vector3dD i("i", 3, { 1, 0, 0 }), j("j", 3, { 0, 1, 0 }), k("k", 3, { 0, 0, 1 });
        vector3dD w(3 * i + 4 * j - 2 * k);
        show_vect(u);
        show_vect(v);
        show_vect(i);
        show_vect(j);
        show_vect(k);
        show_vect(w);
        assert(u == u);
        assert(u != v);
        assert(u + v == v + u);
        assert(u - v == -(v - u));
        assert(-(-u) == u);
        assert(3.0 + u == u + 3.0);
        assert(3.0 * u == u * 3.0);
        assert((u - 3.0) == -(3.0 - u));
        assert((5.0 * u) / 5.0 == u);
        assert(u + vector3dD::zero() == u);
        assert(i.dot(j) == j.dot(k) == k.dot(i) == 0);
        assert(i.cross(j) == k);
        assert(j.cross(k) == i);
        assert(k.cross(i) == j);
        assert(u.cross(v) == -v.cross(u));
        assert(u.cross(v + w) == u.cross(v) + u.cross(w));
        assert((u.cross(v)).dot(u) == 0);
        print(i.angle(j));
        print(M_PI / 2);
        assert(i.angle(j) == M_PI_2);
        assert(j.angle(k) == M_PI_2);
        assert(k.angle(i) == M_PI_2);
        vector3dD uhat = u / u.magnitude(); // unit vector in u direction
        show_vect(u);
        show_vect(uhat);
        print(uhat.magnitude());
        assert(uhat.magnitude() - 1.0 < 1.0e-10);
        print("...test vectors assertions passed");
        print("==================== FINISHED testing vectors ========================");
}

void test_matrices() {
        print("\n==================== TESTING MATRICES ========================");
        matrix3dD a("a", 3, { 3, 2, 0, 0, 0, 1, 2, -2, 1 });
        matrix3dD b("b", 3, { 1, 0, 5, 2, 1, 6, 3, 4, 0 });
        matrix3dD ainv = a.inverse();
        matrix3dD binv = b.inverse();
        print(a);
```

```
        print(b);
        print(ainv);
        print(binv);
        print(a * ainv);
        print(b * binv);
        assert(a * ainv == matrix3dD::identity(3));
        assert(a * ainv == ainv * a);
        assert(b * binv == matrix3dD::identity(3));
        assert(b * binv == binv * b);
        assert(a.transpose().transpose() == a);
        assert(a.transpose().determinant() == a.determinant());
        assert(a + b == b + a);
        assert(a - b == -(b - a));
        assert(3.0 + a == a + 3.0);
        assert(3.0 * a == a * 3.0);
        assert((a + 3.0) - 3.0 == a);
        assert((3.0 * a) / 3.0 == a);
        assert(-(-a) == a);
        matrix3dD zerod("zerod", 3, { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
        assert(zerod.determinant() == 0);
        print("...test matrices assertions passed");
        print("==================== FINISHED testing matrices ========================");
}

void test_matrices_and_vectors() {
        print("\n==================== TESTING MATRICES and VECTORS ========================");
        vector3dD p("p", 2, { 1, 2 });
        matrix3dD m("m", 2, { 1, 2, 3, 4 });
        show_vect(p);
        show_mat(m);
        assert(p * m == m * p);
        vector3dD q("q", 3, { 1, 2, 3 });
        matrix3dD n("n", 3, { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
        show_vect(q);
        show_mat(n);
        assert(q * n == n * q);
        print("...test_matrices_and_vectors assertions passed");
        print("==================== FINISHED testing matrices and vectors ========================");
}

int main(){
        test_vectors();
        test_matrices();
        test_matrices_and_vectors();
        print("... program completed...\n");
        return 0;
}

//============================================================
// end of file: main.cpp
//============================================================
```

```
==================== TESTING VECTORS ========================

u is: <'u', 1 2 4 0>

v is: <'v', 8 16 32 0>

i is: <'i', 1 0 0 0>

j is: <'j', 0 1 0 0>

k is: <'k', 0 0 1 0>

3.000000*i+4.000000*j-2.000000*k is: <'3.000000*i+4.000000*j-2.000000*k', 3 4 -2 0>

1.5708

1.5708

u is: <'u', 1 2 4 0>

u/4.582576 is: <'u/4.582576', 0.218218 0.436436 0.872872 0>

1

...test vectors assertions passed

==================== FINISHED testing vectors ========================


==================== TESTING MATRICES ========================

<'a', <'col0', 3 2 0 0><'col1', 0 0 1 0><'col2', 2 -2 1 0>> OR by rows...

3 0 2

2 0 -2

0 1 1

>

<'b', <'col0', 1 0 5 0><'col1', 2 1 6 0><'col2', 3 4 0 0>> OR by rows...

1 2 3

0 1 4

5 6 0

>

<'Transpose(Cofactor(a))/10.000000', <'col0/10.000000', 0.2 -0.2 0.2 0><'col1/10.000000', 0.2 0.3 -0.3 0><'col2/10.000000', -0 1 0 0>> OR by rows...

0.2 0.2 -0

-0.2 0.3 1

0.2 -0.3 0

>

<'Transpose(Cofactor(b))/1.000000', <'col0/1.000000', -24 20 -5 0><'col1/1.000000', 18 -15 4 0><'col2/1.000000', 5 -4 1 0>> OR by rows...

-24 18 5

20 -15 -4

-5 4 1

>
```

<'a*Transpose(Cofactor(a))/10.000000', <'col0', 1 0 0 0><'col1', 1.11022e-16 1 0 0><'col2', 0 0 1 0>> OR by rows...

1 1.11022e-16 0

0 1 0

0 0 1

>

<'b*Transpose(Cofactor(b))/1.000000', <'col0', 1 0 0 0><'col1', 0 1 0 0><'col2', 0 0 1 0>> OR by rows...

1 0 0

0 1 0

0 0 1

>

...test matrices assertions passed

========== FINISHED testing matrices ==========

========== TESTING MATRICES and VECTORS ==========

p is: <'p', 1 2 -9.25596e+61>

m is: <'m', <'col0', 1 2 3 0><'col1', 4 0 0 0><'col2', 0 0 0 0>> OR by rows...

1 4 0

2 0 0

3 0 0

>

q is: <'q', 1 2 3 0>

n is: <'n', <'col0', 1 2 3 0><'col1', 4 5 6 0><'col2', 7 8 9 0>> OR by rows...

1 4 7

2 5 8

3 6 9

>

...test_matrices_and_vectors assertions passed

========== FINISHED testing matrices and vectors ==========

... program completed...

Press any key to continue . . .