Swinburne University of Technology

School of Science, Computing and Engineering Technologies

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Assignment number and title:	1, Solution Design in C++
Due date:	Sunday, March 30, 2025, 23:59
Lecturer:	Dr. Markus Lumpe

Your name:____________________________ **Your student ID:**_______________

Marker's comments:

Problem	Marks	Obtained
1	38	
2	170	
Total	208	

Extension certification:

This assignment has been given an extension and is now due on __________________

Signature of Convener:_________________________

Problem Set 1: Solution Design in C++

Problem 1

Please start with the solution of tutorial 3 in which we implemented the class `Vector3D`.

In this problem, we want to extend `Vector3D` with two additional features: `Vector3D` equivalence and `Vector3D` textual representation. The former requires defining the **`operator==`**, while the latter can be realized by creating a `toString()` method.

To support the new member function, class `Vector3D` is extended as follows

```
#pragma once

#include "Vector2D.h"

#include <string>
#include <limits>

constexpr float eps = std::numeric_limits<float>::epsilon();

class Vector3D
{
private:

  Vector2D fBaseVector;
  float fW;

public:

  Vector3D( float aX = 1.0f, float aY = 0.0f, float aW = 1.0f ) noexcept;
  Vector3D( const Vector2D& aVector ) noexcept;

  float x() const noexcept { return fBaseVector.x(); }
  float y() const noexcept { return fBaseVector.y(); }
  float w() const noexcept { return fW; }

  float operator[]( size_t aIndex ) const;

  explicit operator Vector2D() const noexcept;

  Vector3D operator*( const float aScalar ) const noexcept;
  Vector3D operator+( const Vector3D& aOther ) const noexcept;
  float dot( const Vector3D& aOther ) const noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream, const Vector3D& aVector );

  // Problem Set 1 extension

  bool operator==( const Vector3D& aOther ) const noexcept;

  std::string toString() const noexcept;
};
```

The **`operator==`** [22 marks] must mutually compare all components of two vectors. The left-hand side of the comparison is **`this`**-object. The right-hand side is the argument passed to **`operator==`**. The argument is a constant reference to another `Vector3D` object. Its value can only be read and not changed. In addition, by using a reference to `Vector3D` we avoid unnecessary copies. The **`operator==`** is also marked const to indicate that it does not modify **`this`**-object.

Comparing floating point values for equality is generally not recommended due to rounding. We often use a trick to achieve this. Rather than comparing for equality, we can compute the absolute difference of two floating point values and return true if the absolute difference is smaller than a predefined epsilon value, the smallest difference between two adjacent floating point numbers. This approach is not without problems, but, in this assignment, we experiment with it and use `std::numeric_limits<float>::epsilon()` for this purpose.

The method `toString()` [16 marks] has to return a textual representation of a 3D vector. For example, `toString()` applied to `Vector3D( 1.0f, 2.0f, 3.0f )` has to yield a string "`[1,2,3]`" as textual representation.

Use `std::stringstream` to implement the `toString()` method. The class `std::stringstream` provides a memory stream. You can use formatted output (i.e., the operator <<) to send data to this stream and at the end, use the method `str()` to obtain the resulting string that `toString()` has to return.

Do not edit the provided files. To implement the required features, create a new source file, say `Vector3D_PS1.cpp`, and define the new feature here. It is not strictly required, but it helps to separate the definitions from the provided code. You must include `Vector3D.h` in the newly created source file to compile.

The file `Main.cpp` contains a test function to check your implementation of the new matrix features. It compiles when C++20 is enabled. The code sequence

```
void runP1()
{
  gCount++;

  constexpr float pi = std::numbers::pi_v<float>;

  Vector3D a( 1.0f, 2.0f, 3.0f );
  Vector3D aa( 1.00000003f, 2.00000008f, 3.00000005f );
  Vector3D b( pi, pi ,pi );
  Vector3D c( 1.23456789f, 9.876543210f, 12435.0987654321f );

  std::cout << "a == aa: " << (a == aa ? "true" : "false") << std::endl;
  std::cout << "a == b: " << (a == b ? "true" : "false") << std::endl;
  std::cout << "a == c: " << (a == c ? "true" : "false") << std::endl;
  std::cout << "b == c: " << (b == c ? "true" : "false") << std::endl;
  std::cout << "a == a: " << (a == a ? "true" : "false") << std::endl;
  std::cout << "b == b: " << (b == b ? "true" : "false") << std::endl;
  std::cout << "c == c: " << (c == c ? "true" : "false") << std::endl;

  std::cout << "Vector aa: " << aa.toString() << std::endl;
  std::cout << "Vector a: " << a.toString() << std::endl;
  std::cout << "Vector b: " << b.toString() << std::endl;
  std::cout << "Vector c: " << c.toString() << std::endl;
}
```

Should produce the following output

```
a == aa: true
a == b: false
a == c: false
b == c: false
a == a: true
b == b: true
c == c: true
Vector aa: [1,2,3]
Vector a: [1,2,3]
Vector b: [3.14159,3.14159,3.14159]
Vector c: [1.23457,9.87654,12435.1]
1 Test(s) completed.
```

Floating point values are printed with standard precision for type **float**. In `Main.cpp`, uncomment the line `#define P1` for this test to work.

Problem 2

Please start with the solution of tutorial 3 in which we implemented the classes `Vector3D` and `Matrix3x3` to perform vector transformations in 2D. This problem also requires the features defined in Problem 1 to be available.

In this problem, we wish to extend the definition of class `Matrix3x3` with some additional matrix operations. In particular, we extend class `Matrix3x3` with

- Matrix equivalence [16]:

  We do not use the algebraic definition here, but rather employ a programmatic one. Two matrices are equivalent (expressed via **`operator==`**) when their mutual respective row vectors are the same.

  Recall the idiom

```
const Matrix3x3& M = *this;
```

  that we used in tutorial 3. The type `Matrix3x3` defines an index operator that returns a constant reference to a row vector. For instance, using the above declaration, we can write `M[0]` rather than `(*this)[0]` to obtain the first row of the matrix represented by **`this`**-object. This approach makes the code more readable and does not incur any runtime overhead.

- Matrix multiplication [50 marks]:

  Two matrices **F** and **G** can be multiplied, provided that the number of columns in **F** is equal to the number of rows in **G**. If **F** is n x m matrix and **G** is an m x p matrix, then the product **FG** is an n x p matrix whose (i, j) entry is given by

$$(\mathbf{FG})_{ij} = \sum_{k=1}^{m} F_{ik} G_{kj}$$

  The entry $(\mathbf{FG})_{ij}$ is the dot product of *row*(**F**, i) and *column*(**G**, j).

  In the implementation, a column vector must be copied at most once via a call to column(). You can declare local variables. Computing the result does not require loops. In addition, recall the idiom

```
const Matrix3x3& M = *this;
```

  that we used in tutorial 3. It is extremely helpful in computing the result matrix. Construct new `Vector3D` objects, the row vectors of the result matrix. Do not create temporaries for the row vectors or the result matrix as it incurs runtime overhead.

- The transpose of a matrix [8 marks]:

  The transpose of an n x m matrix **M**, denoted by $\mathbf{M}^T$, is an m x n matrix for which the (i,j) entry equals M_{ji}. The transpose of

$$\mathbf{M}_{3x3} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \text{ is } \mathbf{M}_{3x3}^T = \begin{bmatrix} M_{11} & M_{21} & M_{31} \\ M_{12} & M_{22} & M_{32} \\ M_{13} & M_{23} & M_{33} \end{bmatrix}.$$

  In the implementation, every column must be accessed once via calls to column(). Avoid temporaries.

- Determinant of a matrix [26 marks]:

  The determinant is a scalar value distilled as a function of the entities of a square matrix. It characterizes some important properties of a square matrix that allow us to test, for example, if the matrix is invertible or if the matrix is a rotation matrix.

  For a 3 x 3 matrix **M**, the determinant of **M** is given by

$$\begin{aligned} \det \mathbf{M} &= \quad M_{11}(M_{22}M_{33} - M_{23}M_{32}) \\ &\quad - M_{12}(M_{21}M_{33} - M_{23}M_{31}) \\ &\quad + M_{13}(M_{21}M_{32} - M_{22}M_{31}) \end{aligned}$$

  Computing the result does not require loops. In addition, recall the idiom

```
const Matrix3x3& M = *this;
```

  Using the constant reference `M`, you can naturally express the determinant without copying data. The row vectors are of type `Vector3D` which provides an index operator for accessing the corresponding column entry.

- A test whether a matrix **M** is invertible [4 marks]:
  A matrix is invertible if its determinant is not zero. The function does not trigger an exception.
- The inverse of a matrix [52 marks]:

  The inverse of a matrix allows us to represent division of matrices, a concept not defined for matrices. Technically, the inverse of a matrix is a multidimensional generalization of the reciprocal of a number: the product of a number and its reciprocal is 1. The product of a matrix **M** with its inverse $\mathbf{M}^{-1}$ is the identity matrix **I**: $\mathbf{M}\mathbf{M}^{-1} = \mathbf{I}$.

  For a 3 x 3 matrix **M**, the inverse matrix $\mathbf{M}^{-1}$ is given by

$$\mathbf{M}^{-1} = \frac{1}{\det \mathbf{M}} \begin{bmatrix} M_{22}M_{33} - M_{23}M_{32} & M_{13}M_{32} - M_{12}M_{33} & M_{12}M_{23} - M_{13}M_{22} \\ M_{23}M_{31} - M_{21}M_{33} & M_{11}M_{33} - M_{13}M_{31} & M_{13}M_{21} - M_{11}M_{23} \\ M_{21}M_{32} - M_{22}M_{31} & M_{12}M_{31} - M_{11}M_{32} & M_{11}M_{22} - M_{12}M_{21} \end{bmatrix}$$

  Please note that the determinant of the matrix M occurs as a denominator on the right-hand side. It cannot be zero. A matrix has an inverse if its determinant is not zero. This requirement must be guaranteed via assertion checking.

  Computing the result does not require loops. In addition, recall the idiom

```
const Matrix3x3& M = *this;
```

  Using the constant reference `M`, you can naturally express the inverse without copying data. The row vectors are of type `Vector3D` which provides an index operator for accessing the corresponding column entry.

  Use the given formula for calculation. It is the explicitly derived formula commonly used in computer graphics.
- Output operator for `Matrix3x3` [14 marks]:
  We can rely on the newly defined `toString()` method in `Vector3D` for this purpose.

To accommodate these operations, we extend class `Matrix3x3` as follows

```
#pragma once

#include "Vector3D.h"

class Matrix3x3
{
private:

  Vector3D fRows[3];

public:

  Matrix3x3() noexcept;
  Matrix3x3( const Vector3D& aRow1, const Vector3D& aRow2, const Vector3D& aRow3 ) noexcept;

  Matrix3x3 operator*( const float aScalar ) const noexcept;
  Matrix3x3 operator+( const Matrix3x3& aOther ) const noexcept;

  Vector3D operator*( const Vector3D& aVector ) const noexcept;

  static Matrix3x3 getS( const float aX = 1.0f, const float aY = 1.0f ) noexcept;
  static Matrix3x3 getT( const float aX = 0.0f, const float aY = 0.0f ) noexcept;
  static Matrix3x3 getR( const float aAngleInDegree = 0.0f ) noexcept;

  const Vector3D& row( size_t aRowIndex ) const noexcept;
  const Vector3D column( size_t aColumnIndex ) const noexcept;

  const Vector3D& operator[]( size_t aRowIndex ) const noexcept;

  // Problem Set 1 features

  bool operator==( const Matrix3x3& aOther ) const noexcept;

  Matrix3x3 operator*( const Matrix3x3& aOther ) const noexcept;

  Matrix3x3 transpose() const noexcept;

  float det() const noexcept;
  bool hasInverse() const noexcept;
  Matrix3x3 inverse() const noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream, const Matrix3x3& aMatrix );
};
```

Do not edit the provided files. To implement the required features, create a new source file, say `Matrix3x3_PS1.cpp`, and define the new features here. This approach helps separating your definitions from the provided code. You must include `Matrix3x3.h` in the newly created source file to compile.

The file `Main.cpp` contains a test function to check your implementation of the new matrix features. The code sequence

```
void runP2()
{
  gCount++;

  Matrix3x3 I;

  Matrix3x3 M1 = Matrix3x3::getR( 45.0f );
  Matrix3x3 M1T = M1.transpose();
  Matrix3x3 Prod = M1 * M1T;

  if ( Prod == I )
  {
    std::cout << "Matrix M1 is a rotation matrix." << std::endl;
  }
  else
  {
    std::cout << "Error." << std::endl;
  }

  std::cout << "det M = " << M1.det() << std::endl;
```

```
  Matrix3x3 M2 ( Vector3D( 25.0f, -3.0f, -8.0f ),
                 Vector3D( 6.0f, 2.0f, 15.0f ),
                 Vector3D( 11.0f, -3.0f, 4.0f ) );

  std::cout << "Test matrix M2:" << std::endl;
  std::cout << M2 << std::endl;

  // test multiplication

  std::cout << "M2 * M2 = " << std::endl;
  std::cout << M2 * M2 << std::endl;

  // test determinate

  std::cout << "det M2 = " << M2.det() << std::endl;

  // test hasInverse

  std::cout << "Has M2 an inverse? " << (M2.hasInverse() ? "Yes" : "No") << std::endl;

  // test transpose
  std::cout << "transpose of M2:" << std::endl;
  std::cout << M2.transpose() << std::endl;

  // test inverse
  std::cout << "inverse of M2:" << std::endl;
  std::cout << M2.inverse() << std::endl;

  std::cout << "inverse of M2 * 45:" << std::endl;
  std::cout << M2.inverse() * 45.0f << std::endl;
}
```

Should produce the following output

```
Matrix M1 is a rotation matrix.
det M = 1
Test matrix M2:
[[25,-3,-8],[6,2,15],[11,-3,4]]
M2 * M2 =
[[519,-57,-277],[327,-59,42],[301,-51,-117]]
det M2 = 1222
Does M2 have an inverse? Yes
transpose of M2:
[[25,6,11],[-3,2,-3],[-8,15,4]]
inverse of M2:
[[0.0433715,0.0294599,-0.0237316],[0.115385,0.153846,-0.346154],
[-0.0327332,0.0343699,0.0556465]]
inverse of M2 * 45:
[[1.95172,1.3257,-1.06792],[5.19231,6.92308,-15.5769],[-1.473,1.54664,2.50409]]
1 Test(s) completed.
```

Floating point values are printed with standard precision for type **`float`**. In `Main.cpp`, uncomment the line **`#define`** `P2` for this test to work.

Submission deadline: Sunday, March 30, 2025, 23:59.

Submission procedure: Follow the instructions on Canvas. Submit electronically the PDF of the printed code for `Vector3D_PS1.cpp` and `Matrix3x3_PS1.cpp`. Upload the sources of `Vector3D_PS1.cpp` and `Matrix3x3_PS1.cpp` to Canvas.

The sources need to compile in the presence of the solution artifacts provided on Canvas.