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:	Wednesday, March 27, 2024, 23:59
Lecturer:	Dr. Markus Lumpe

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

Marker's comments:

Problem	Marks	Obtained
1	26	
2	98	
3	32	
Total	156	

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

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

In this problem, we which to extend `Vector3D` with a `toString()` method. This method 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.

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

```
#pragma once

#include "Vector2D.h"

#include <string>

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

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

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 have to include `Vector3D.h` in the new source for the code to compile.

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.

Numerical data must be rounded to 10^{-4}, that is, four positions after the period.

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

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

  Vector3D a( 1.0f, 2.0f, 3.0f );
  Vector3D b(static_cast<float>(M_PI),static_cast<float>(M_PI),static_cast<float>(M_PI));
  Vector3D c(1.23456789f, 9.876543210f, 12435.0987654321f);

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

Should produce the following output

```
Vector a: [1,2,3]

Vector b: [3.1416,3.1416,3.1416]

Vector c: [1.2346,9.8765,12435.1]
```

Floating point values are printed with standard precision for type **`float`**.

Problem 2

Start with the solution of tutorial 3 in which we implemented the classes `Vector3D` and `Matrix3x3` to perform vector transformations in 2D.

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 multiplication [26 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, every row and column must be accessed once via calls to row() and column(). You can declare local variables. Computing the result does not require loops.

- Determinate of a matrix [22 marks]:
  The determinate is a scalar value that is a function of the entities of a square matrix. It characterizes some properties of a square matrix, for example, if the matrix is invertible or if the matrix is a rotation matrix.
  For a 3 x 3 matrix **M**, the determinate of **M** is given by

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

  In the implementation, every row must be accessed once via calls to row(). You can declare local variables. Computing the result does not require loops. The row vectors are of type `Vector3D` which provides an index operator to access the corresponding column entry.

- 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 is equal to M_{ji}. For

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

  In the implementation, every column must be accessed once via calls to column(). You may use local variables, but it is not strictly necessary.

- 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 [24 marks]:
  The inverse of a matrix, if it exists, allows us to represent division of matrices, a concept that is not defined for matrices. Technically, the inverse of a matrix is a

multidimensional generalization of the concept of 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**: $\mathbf{MM}^{-1} = \mathbf{I}$.
For a 3 x 3 matrix **M**, the inverse matrix $\mathbf{M}^{-1}$ is given by

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

In the implementation, every row must be accessed once via calls to row(). You can declare local variables.
The implementation would need to compute the determinate of **M** and verify that it is not zero. Use the given formula for calculation.

- Output operator for `Matrix3x3` [14]:
  We can rely on the newly defined `toString()` method in `Vector3D` for this purpose.

To accommodate these operations, class `Matrix3x3` is extended as follows

```
#pragma once

#include <ostream>

#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 scale( const float aX = 1.0f, const float aY = 1.0f );
  static Matrix3x3 translate( const float aX = 0.0f, const float aY = 0.0f );
  static Matrix3x3 rotate( const float aAngleInDegree = 0.0f );

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

  // Problem Set 1 features

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

  float det() const noexcept;
  Matrix3x3 transpose() const noexcept;

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

  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. It is not strictly required, but it helps to separate the definitions from the provided code. You have to include `Matrix3x3.h` in the new source for the code 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 M ( Vector3D( 25.0f, -3.0f, -8.0f ),
                Vector3D( 6.0f, 2.0f, 15.0f ),
                Vector3D( 11.0f, -3.0f, 4.0f ) );

  cout << "Test matrix M:" << endl;
  cout << M << endl;

  // test multiplication

  cout << "M * M = " << endl;
  cout << M * M << endl;

  // test determinate

  cout << "det M = " << M.det() << endl;

  // test hasInverse

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

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

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

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

Should produce the following output

```
Test matrix M:
[[25,-3,-8],[6,2,15],[11,-3,4]]
M * M =
[[519,-57,-277],[327,-59,42],[301,-51,-117]]
det M = 1222
Has M an inverse? Yes
transpose of M:
[[25,6,11],[-3,2,-3],[-8,15,4]]
inverse of M:
[[0.0434,0.0295,-0.0237],[0.1154,0.1538,-0.3462],[-0.0327,0.0344,0.0556]]
inverse of M * 45:
[[1.9517,1.3257,-1.0679],[5.1923,6.9231,-15.5769],[-1.473,1.5466,2.5041]]
```

Floating point values are printed with standard precision for type **`float`**.

Problem 3

Carl Friedrich Gauss and Carl Gustav Jacob Jacobi invented the trapezoid formula to calculate the area of a trapezoid in the 18th century. We can use the trapezoid formula to determine the both the area of a polygon and the orientation of the vertices of the polygon. If the vertices are ordered in clockwise order, then the area is negative. If the vertices are arranged in counterclockwise order, then the area is positive as shown in Figure 1. In computer graphics, we use the orientation of a polygon to determine whether the polygon faces the viewer or not. The latter allows for a process called *back face culling* that means, we do not have to draw the polygon. Only if the polygon faces the viewer (even partly) do we have to draw it.

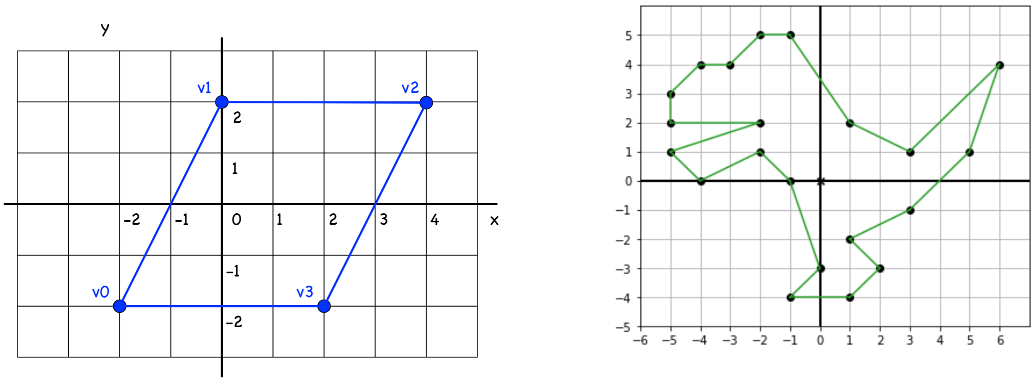

Figure 1: Parallelogram with signed area -16, dinosaur with signed area 38.5.

The trapezoid formula is given below

$$A = \frac{1}{2}\sum_{i=1}^{n}(y_i + y_{i\oplus 1})(x_i - x_{i\oplus i})$$

It sums up a sequence of oriented areas of trapezoids with vertices v_i and v_{i+1} as one of its four edges. Please note that we need to connect the last with the first vertex. In the trapezoid formula, this is expressed by the modulus operator (i.e., $i\oplus 1$). Naturally, it is always possible to avoid the use of the expensive modulus operation by separating the logic into two parts (i.e., we explicitly define the connection between the last and the first vertex).

In order the add support for the signed area calculation, start with the solution of tutorial 2 (see Canvas) and use the extended specification of class `Polygon` is shown below. Please note that the extended version also includes a method to transform a given polygon. Polygon transformation requires a 3 x 3 matrix which is multiplied with every vertex of the polygon.

```
#pragma once

#include "Vector2D.h"
#include "Matrix3x3.h"

constexpr size_t MAX_VERTICES = 30;

class Polygon
{
private:
  Vector2D fVertices[MAX_VERTICES];
  size_t fNumberOfVertices;

public:

  Polygon() noexcept;

  void readData( std::istream& aIStream );

  size_t getNumberOfVertices() const noexcept;
  const Vector2D& getVertex( size_t aIndex ) const;
  float getPerimeter() const noexcept;

  Polygon scale( float aScalar ) const noexcept;

  // Problem Set 1 extension

  float getSignedArea() const noexcept;

  Polygon transform( const Matrix3x3& aMatrix ) const noexcept;
};
```

As in 0, do not edit the provided files. Rather create a new source file, called `Polygon_PS1.cpp`, to implement the new methods `getSignedArea()` and `transform()`.

There are two sample files that you can use to test your solution: `Parallelogram.txt` and `Data.txt`. The file `Main.cpp` contains a test function, `testProblem2()`, to verify your implementation. For the input file `Parallelogram.txt` the test function outputs

```
Signed area: -16
Signed area of rotated polygon: -16
Polygon transformation successful.
```

The vertices in the parallelogram are arranged in clockwise order. The area of a polygon does not change when it is simply rotated around the origin.

For the input file `Data.txt` the test function outputs

```
Signed area: 38.5
Signed area of rotated polygon: 38.5
Polygon transformation successful.
```

The vertices in the T-Rex polygon are arranged in counterclockwise order.

Submission deadline: Monday, March 27, 2023, 10:30.

Submission procedure: Follow the instruction on Canvas. Submit electronically the PDF of the printed code for `Vector3D_PS1.cpp`, `Matrix3x3_PS1.cpp`, and `PolygonPS1.cpp`. Upload the sources of `Vector3D_PS1.cpp`, `Matrix3x3_PS1.cpp`, and `PolygonPS1.cpp` to Canvas.

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