Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 1, Solution Design in C++
Due date: Thursday, March 24, 2022, 14:30
Lecturer: Dr. Markus Lumpe

Your name: Ta Quang Tung **Your student ID:** 104222196

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Obtained
1	38	
2	60	
3	38	
4	20	
Total	156	

Extension certification:

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

Signature of Convener:_____

PolygonPS1.cpp

```cpp
#include <iostream>
#include "Polygon.h"

// Constructor for the Polygon class.
// Initializes the number of vertices to 0.
Polygon::Polygon() : fNumberOfVertices(0) {}

// Getter for the number of vertices.
size_t Polygon::getNumberOfVertices() const
{
    return fNumberOfVertices;
}

// Getter for a specific vertex of the polygon.
const Vector2D &Polygon::getVertex(size_t index) const
{
    if (index < fNumberOfVertices)
    {
        return fVertices[index];
    }

    throw std::out_of_range("Index out of range.");
}

// Reads the polygon data from an input stream.
void Polygon::readData(std::istream &aIstream)
{
    while (aIstream >> fVertices[fNumberOfVertices])
    {
        fNumberOfVertices++;
    }
}

// Calculates the perimeter of the polygon.
// The perimeter is the sum of all the polygon's sides.
// Each side is made up of two consecutive vertices.
// Note: The first and last vertices in the array make up a side.
float Polygon::getPerimeter() const
{
    float perimeter = 0.0f;

    for (int i = 0; i < fNumberOfVertices; i++)
    {
        perimeter += (fVertices[(i + 1) % fNumberOfVertices] - fVertices[i]).length();
    }

    return perimeter;
}

// Scales the polygon by a scalar factor, resulting in a new polygon.
Polygon Polygon::scale(float aScalar)
{
    Polygon result = *this;

    for (int i = 0; i < fNumberOfVertices; i++)
    {
```

```
        result.fVertices[i] = result.fVertices[i] * aScalar;
    }

    return result;
}

// Calculates the signed area of the polygon using the shoelace algorithm.
float Polygon::getSignedArea() const
{
    float signedArea = 0.0f, determinant;

    for (int i = 0; i < fNumberOfVertices - 1; i++)
    {
        determinant = fVertices[i].getX() * fVertices[i + 1].getY() - fVertices[i].getY() * fVertices[i + 1].getX();
        signedArea += determinant;
    }

    determinant = fVertices[fNumberOfVertices - 1].getX() * fVertices[0].getY() - fVertices[fNumberOfVertices - 1].getY() * fVertices[0].getX();
    signedArea += determinant;

    signedArea *= 0.5;

    return signedArea;
}
```

PolynomialPS1.cpp

```cpp
#include <iostream>
#include <cmath>

#include "Polynomial.h"

// Constructor for the Polynomial class.
// Initializes the degree and all coefficients to 0.
Polynomial::Polynomial() : fDegree(0)
{
    for (size_t i = 0; i < MAX_DEGREE + 1; i++)
    {
        fCoeffs[i] = 0.0;
    }
}

// Reads the degree of the polynomial first, then all the coefficients.
// The number of coefficients = the degree + 1.
std::istream &operator>>(std::istream &aIStream, Polynomial &aObject)
{
    aIStream >> aObject.fDegree;

    for (size_t i = 0; i <= aObject.fDegree; i++)
    {
        aIStream >> aObject.fCoeffs[i];
    }

    return aIStream;
}

// Prints the polynomial, ignoring the 0.0 coefficients.
std::ostream &operator<<(std::ostream &aOStream, const Polynomial &aObject)
{
    for (size_t i = 0; i <= aObject.fDegree; i++)
    {
        if (aObject.fCoeffs[i] != 0.0)
        {
            if (i != 0)
            {
                aOStream << " + ";
            }

            aOStream << aObject.fCoeffs[i] << "x^" << aObject.fDegree - i;
        }
    }

    return aOStream;
}

// Multiplies the two polynomials of degrees a, b to produce a new polynomial of degree a + b.
// Procedure description:
// Given polynomial A of degree a, the coefficient at index i (0 <= i <= a) corresponds to the term of degree a - i.
//       polynomial B of degree b, the coefficient at index j (0 <= j <= b) corresponds to the term of degree b - j.
// Let polynomial C be the product of A and B. Therefore, its degree is a + b.
```

```cpp
// Multiplying A by B means multiplying each term of A by each term of B then summing all the pairwise products.
// Each pairwise product is a term with coefficient A.coeff[i] * B.coeff[j] and degree (a - i) + (b - j).
// This degree corresponds to index (a + b) - [(a - i) + (b - j)] = i + j in the coefficient array.
// Since some pairwise products may share the same degree, their coefficients must be added at the end of the process.
Polynomial Polynomial::operator*(const Polynomial &aRHS) const
{
    Polynomial result;
    result.fDegree = fDegree + aRHS.fDegree;

    size_t i, j;

    for (i = 0; i <= fDegree; i++)
    {
        for (j = 0; j <= aRHS.fDegree; j++)
        {
            result.fCoeffs[i + j] += fCoeffs[i] * aRHS.fCoeffs[j];
        }
    }

    return result;
}

// Compares the two polynomials, first by their degrees then by their coefficients.
bool Polynomial::operator==(const Polynomial &aRHS) const
{
    if (fDegree != aRHS.fDegree)
        return false;

    for (size_t i = 0; i <= fDegree; i++)
    {
        if (fCoeffs[i] != aRHS.fCoeffs[i])
        {
            return false;
        }
    }

    return true;
}

// Calculates the value of the polynomial for a given X.
double Polynomial::operator()(double aX) const
{
    double result = 0.0;

    for (size_t i = 0; i <= fDegree; i++)
    {
        result += fCoeffs[i] * pow(aX, fDegree - i);
    }

    return result;
}

// Calculates the derivative as a new polynomial with degree fDegree - 1.
Polynomial Polynomial::getDerivative() const
{
    Polynomial derivative;

```

```
    if (fDegree == 0)
    {
        derivative.fDegree = 0;
    }
    else
    {
        derivative.fDegree = fDegree - 1;
    }

    for (size_t i = 0; i <= derivative.fDegree; i++)
    {
        derivative.fCoeffs[i] = fCoeffs[i] * (fDegree - i);
    }

    return derivative;
}

// Calculates the indefinite integral as a new polynomial with degree fDegree + 1.
Polynomial Polynomial::getIndefiniteIntegral() const
{
    Polynomial integral;

    integral.fDegree = fDegree + 1;

    for (size_t i = 0; i <= fDegree; i++)
    {
        integral.fCoeffs[i] = fCoeffs[i] / (fDegree - i + 1);
    }

    return integral;
}

// Calculates the definite integral on the interval [aXLow, aXHigh].
double Polynomial::getDefiniteIntegral(double aXLow, double aXHigh) const
{
    Polynomial indefiniteIntegral = getIndefiniteIntegral();
    return indefiniteIntegral(aXHigh) - indefiniteIntegral(aXLow);
}
```

Combination.cpp

```cpp
#include "Combination.h"

// Constructor for the Combination class.
// Initializes the values of N and K.
Combination::Combination(size_t aN, size_t aK) : fN(aN), fK(aK) { }

// Getter function for fN.
size_t Combination::getN() const {
    return fN;
}

// Getter function for fK.
size_t Combination::getK() const {
    return fK;
}

// Calculates the result of N choose K.
// Procedure description:
//  n    (n-0)   (n-1)         (n - (k - 1))
// ( ) = ----- * ----- * ... * -------------
//  k      1       2                 k
unsigned long long Combination::operator()() const {
    unsigned long long numerator = 1, denominator = 1;

    for (size_t i = 0; i < fK; i++) {
        numerator *= fN - i;
        denominator *= i + 1;
    }

    return numerator / denominator;
}
```

BernsteinBasisPolynomial.cpp

```cpp
#include <cmath>
#include "BernsteinBasisPolynomial.h"

// Constructor for the BernsteinBasisPolynomial class.
// Initializes the Combination factor fFactor with the supplied V and N values.
BernsteinBasisPolynomial::BernsteinBasisPolynomial(unsigned int aV, unsigned int aN) : fFactor(Combination(aN, aV)) {}

// Calculates the result of the polynomial for a given value of X.
// Procedure description:
// Output = nCv * x^v * (1-x)^(n-v)
// Note: Due to the implementation of Combination, v is denoted k instead.
double BernsteinBasisPolynomial::operator()(double aX) const
{
    return fFactor() * pow(aX, fFactor.getK()) * pow(1 - aX, fFactor.getN() - fFactor.getK());
}
```