

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: Sunday, October 08, 2023, 23:59 (VN Time)
Lecturer: Dr. Van Dai PHAM

Your name: Nguyen Quoc Bao Huynh **Your student ID:** 103804535

Check Tutorial	Mon 10:00	Tues 10:30	Tues 12:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 10:00 Innovation Lab	Thurs 14:00	Frid 10:00
								X		

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: _____

Problem 1:

PolygonPS1.cpp:

```
1  #include "Polygon.h"
2  #include <stdexcept>
3
4  using namespace std;
5
6  // Constructor
7  Polygon::Polygon()
8  {
9      fNumberOfVertices = 0;
10 }
11
12 // Get the number of vertices in the polygon
13 size_t Polygon::getNumberOfVertices() const
14 {
15     return fNumberOfVertices;
16 }
17
18 // Get a specific vertex of the polygon by index
19 const Vector2D& Polygon::getVertex( size_t aIndex ) const
20 {
21     // Check if the index is within the bounds of the array
22     if ( aIndex < fNumberOfVertices )
23     {
24         return fVertices[aIndex];
25     }
26
27     // Throw an exception if the index is out of range
28     throw out_of_range( "Illegal index value." );
29 }
30
31 // Read the data of the polygon from an input stream
32 void Polygon::readData( istream& aIStream )
33 {
34     // Read vertices from the input stream until it fails or ends
35     while ( aIStream >> fVertices[fNumberOfVertices] )
36     {
37         fNumberOfVertices++;
38     }
39 }
```

```

40
41 // Calculate the perimeter of the polygon
42 float Polygon::getPerimeter() const
43 {
44     float Result = 0.0f;
45
46     // Check if the polygon has more than 2 vertices
47     if ( fNumberOfVertices > 2 )
48     {
49         // Calculate the sum of the lengths of all edges
50         for ( size_t i = 1; i < fNumberOfVertices; i++ )
51         {
52             Result += (fVertices[i] - fVertices[i - 1]).length();
53         }
54
55         // Add the length of the last edge connecting the last and first vertices
56         Result += (fVertices[0] - fVertices[fNumberOfVertices - 1]).length();
57     }
58
59     return Result;
60 }
61
62 // Scale the polygon by a scalar value
63 Polygon Polygon::scale( float aScalar ) const
64 {
65     // Create a copy of the current polygon
66     Polygon Result = *this;
67
68     // Scale each vertex by the scalar value
69     for ( size_t i = 0; i < fNumberOfVertices; i++ )
70     {
71         Result.fVertices[i] = fVertices[i] * aScalar;
72     }
73
74     return Result;
75 }
76
77 float Polygon::getSignedArea() const {
78     float area = 0.0f;

```

```

77 float Polygon::getSignedArea() const {
78     float area = 0.0f;
79
80     // Apply the Shoelace formula
81     for (size_t i = 0; i < fNumberOfVertices; i++) {
82         const Vector2D& currentVertex = fVertices[i];
83         const Vector2D& nextVertex = fVertices[(i + 1) % fNumberOfVertices];
84         area += (currentVertex.getX() * nextVertex.getY()) - (currentVertex.getY() * nextVertex.getX());
85     }
86
87     // Divide the final result by 2 to get the signed area
88     area /= 2.0f;
89
90     return area;
91 }

```

Problem 2:

PolynomialPS1.cpp:

```
1  #include <iostream>
2  #include "Polynomial.h"
3  using namespace std;
4
5  Polynomial::Polynomial() : fDegree(0)
6  {
7      for (size_t i = 0; i <= MAX_DEGREE; i++) {
8          fCoeffs[i] = 0.0;
9      }
10 }
11
12 Polynomial Polynomial::operator*(const Polynomial& aRHS) const {
13     Polynomial multiplyResult;
14     multiplyResult.fDegree = fDegree + aRHS.fDegree;
15
16     // Multiply polynomials
17     for (size_t i = 0; i <= fDegree; i++) {
18         for (size_t j = 0; j <= aRHS.fDegree; j++) {
19             multiplyResult.fCoeffs[i + j] += fCoeffs[i] * aRHS.fCoeffs[j];
20         }
21     }
22
23     return multiplyResult;
24 }
25
26 bool Polynomial::operator==(const Polynomial& aRHS) const
27 {
28     bool Result = fDegree == aRHS.fDegree;
29
30     // Compare coefficients of polynomials
31     for (size_t i = 0; i <= fDegree; i++)
32     {
33         if (fCoeffs[i] != aRHS.fCoeffs[i])
34         {
35             Result = false;
36         }
37     }
38     return Result;
39 }
```

```

40
41 double Polynomial::operator()(double aX) const
42 {
43     double result = 0.0;
44
45     // Evaluate polynomial using Horner's method
46     for (int i = fDegree; i >= 0; i--)
47     {
48         result += fCoeffs[i] * pow(aX, i);
49     }
50
51     return result;
52 }
53
54 Polynomial Polynomial::getDerivative() const
55 {
56     Polynomial derivative;
57     derivative.fDegree = fDegree - 1;
58
59     // Compute the derivative of the polynomial
60     for (int i = fDegree; i >= 1; i--)
61     {
62         derivative.fCoeffs[i - 1] = fCoeffs[i] * i;
63     }
64
65     return derivative;
66 }
67
68 Polynomial Polynomial::getIndefiniteIntegral() const
69 {
70     Polynomial integral;
71     integral.fDegree = fDegree + 1;
72
73     // Compute the indefinite integral of the polynomial
74     for (int i = fDegree + 1; i >= 1; i--)
75     {
76         integral.fCoeffs[i] = fCoeffs[i - 1] / i;
77     }
78

```

```

67
68 Polynomial Polynomial::getIndefiniteIntegral() const
69 {
70     Polynomial integral;
71     integral.fDegree = fDegree + 1;
72
73     // Compute the indefinite integral of the polynomial
74     for (int i = fDegree + 1; i >= 1; i--)
75     {
76         integral.fCoeffs[i] = fCoeffs[i - 1] / i;
77     }
78
79     return integral;
80 }
81
82 double Polynomial::getDefiniteIntegral(double aXLow, double aXHigh) const
83 {
84     Polynomial indefiniteIntegral = getIndefiniteIntegral();
85     double integralLow = 0.0;
86     double integralHigh = 0.0;
87
88     // Compute the definite integral using the indefinite integral
89     for (int i = fDegree + 1; i >= 0; i--)
90     {
91         integralLow += indefiniteIntegral.fCoeffs[i] * pow(aXLow, i);
92         integralHigh += indefiniteIntegral.fCoeffs[i] * pow(aXHigh, i);
93     }
94
95     return integralHigh - integralLow;
96 }
97
98 istream& operator>>(istream& aIStream, Polynomial& aObject) {
99     aIStream >> aObject.fDegree;
100
101     // Read the polynomial coefficients from input
102     for (int i = aObject.fDegree; i >= 0; i--) {
103         aIStream >> aObject.fCoeffs[i];
104     }
105

```

```

100
101 // Read the polynomial coefficients from input
102 for (int i = aObject.fDegree; i >= 0; i--) {
103     aIStream >> aObject.fCoeffs[i];
104 }
105
106 return aIStream;
107 }
108
109 ostream& operator<<(ostream& aOStream, const Polynomial& aObject) {
110     bool lPrint = false;
111
112     // Print the polynomial expression
113     for (int i = aObject.fDegree; i >= 0; i--) {
114         if (aObject.fCoeffs[i] != 0.0) {
115             if (lPrint) {
116                 aOStream << " + ";
117             }
118             else {
119                 lPrint = true;
120             }
121
122             double coeff = abs(aObject.fCoeffs[i]);
123             if (coeff != 1.0 || i == 0)
124                 aOStream << aObject.fCoeffs[i];
125
126             if (i > 0)
127                 aOStream << "x";
128
129             if (i > 1)
130                 aOStream << "^" << i;
131         }
132     }
133
134     return aOStream;
135 }

```

Problem 3:

Combination.cpp:

```
1  #include "Combination.h"
2
3  Combination::Combination(size_t aN, size_t aK)
4      : fN(aN), fK(aK)
5      {
6      }
7
8  size_t Combination::getN() const
9      {
10         return fN;
11     }
12
13  size_t Combination::getK() const
14      {
15         return fK;
16     }
17
18  unsigned long long Combination::operator()() const
19      {
20         unsigned long long result = 1;
21
22         // Calculate the combination using the formula
23         // (n choose k) = n! / (k! * (n - k)!)
24         for (size_t i = 1; i <= fK; ++i)
25         {
26             // Multiply the result by (fN - i + 1)
27             // to compute the numerator of the combination formula
28             result *= fN - i + 1;
29
30             // Divide the result by i
31             // to compute the denominator of the combination formula
32             result /= i;
33         }
34
35         return result;
36     }
```


Problem 4:

BernsteinBasePolynomial.cpp

```
1  #include "BernsteinBasisPolynomial.h"
2  #include <cmath>
3
4  // Constructor for b(v,n)
5  BernsteinBasisPolynomial::BernsteinBasisPolynomial(unsigned int aV, unsigned int aN)
6  : fFactor(aN, aV) {}
7
8  // Operator to calculate Bernstein polynomial
9  double BernsteinBasisPolynomial::operator()(double aX) const {
10     double result = 0.0;
11     unsigned int n = fFactor.getN();
12     unsigned int v = fFactor.getK();
13
14     result = fFactor() * pow(aX, v) * pow(1 - aX, n - v);
15
16     return result;
17 }
```