Problem 1:

PolygonPS1.cpp:

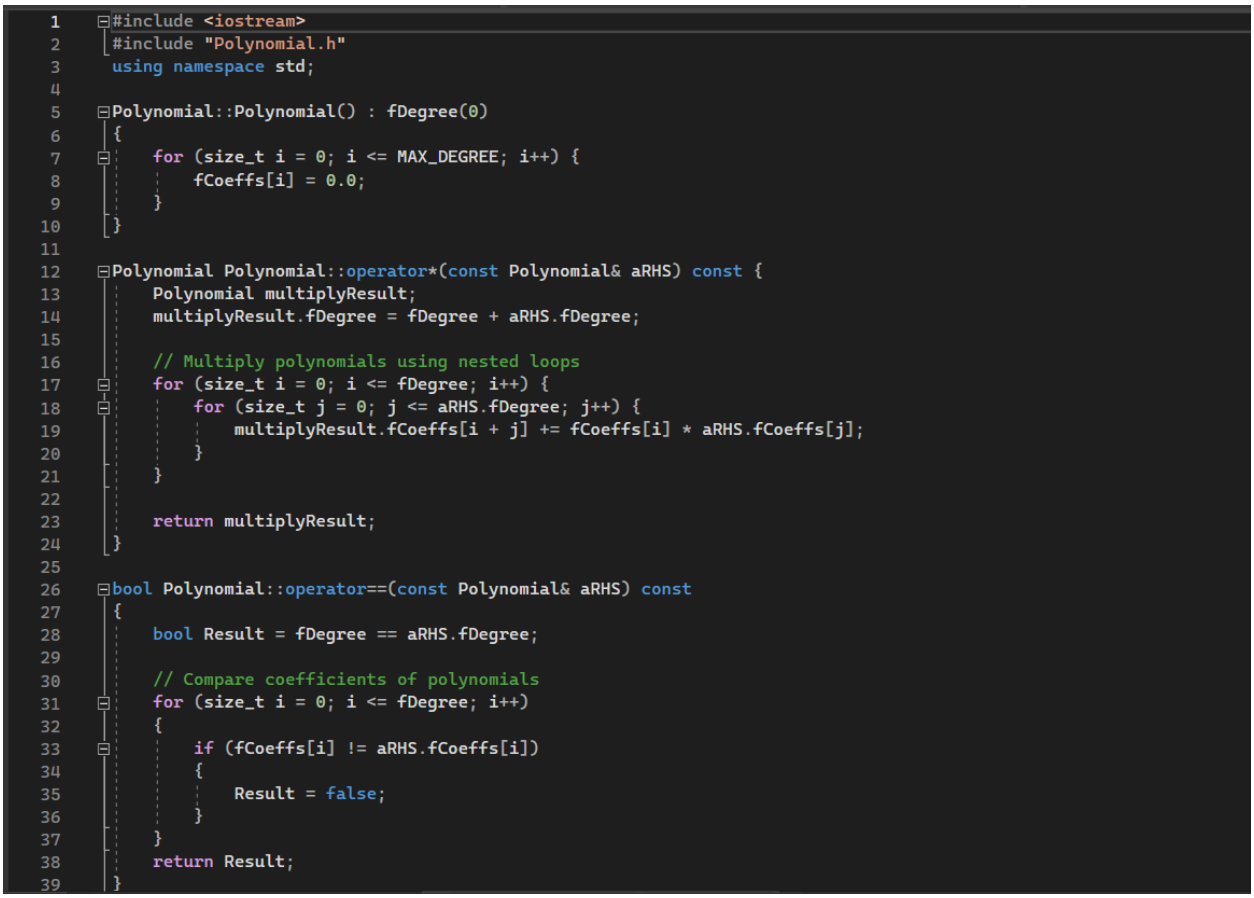
```cpp
#include "Polygon.h"
#include <stdexcept>

using namespace std;

// Constructor
Polygon::Polygon()
{
    fNumberOfVertices = 0;
}

// Get the number of vertices in the polygon
size_t Polygon::getNumberOfVertices() const
{
    return fNumberOfVertices;
}

// Get a specific vertex of the polygon by index
const Vector2D& Polygon::getVertex( size_t aIndex ) const
{
    // Check if the index is within the bounds of the array
    if ( aIndex < fNumberOfVertices )
    {
        return fVertices[aIndex];
    }

    // Throw an exception if the index is out of range
    throw out_of_range( "Illegal index value." );
}

// Read the data of the polygon from an input stream
void Polygon::readData( istream& aIStream )
{
    // Read vertices from the input stream until it fails or ends
    while ( aIStream >> fVertices[fNumberOfVertices] )
    {
        fNumberOfVertices++;
    }
}
```

```cpp

    // Calculate the perimeter of the polygon
float Polygon::getPerimeter() const
{
    float Result = 0.0f;

    // Check if the polygon has more than 2 vertices
    if ( fNumberOfVertices > 2 )
    {
        // Calculate the sum of the lengths of all edges
        for ( size_t i = 1; i < fNumberOfVertices; i++ )
        {
            Result += (fVertices[i] - fVertices[i - 1]).length();
        }

        // Add the length of the last edge connecting the last and first vertices
        Result += (fVertices[0] - fVertices[fNumberOfVertices - 1]).length();
    }

    return Result;
}

    // Scale the polygon by a scalar value
Polygon Polygon::scale( float aScalar ) const
{
    // Create a copy of the current polygon
    Polygon Result = *this;

    // Scale each vertex by the scalar value
    for ( size_t i = 0; i < fNumberOfVertices; i++ )
    {
        Result.fVertices[i] = fVertices[i] * aScalar;
    }

    return Result;
}

float Polygon::getSignedArea() const {
    float area = 0.0f;
```

```cpp
float Polygon::getSignedArea() const {
    float area = 0.0f;

    // Apply the Shoelace formula
    for (size_t i = 0; i < fNumberOfVertices; i++) {
        const Vector2D& currentVertex = fVertices[i];
        const Vector2D& nextVertex = fVertices[(i + 1) % fNumberOfVertices];
        area += (currentVertex.getX() * nextVertex.getY()) - (currentVertex.getY() * nextVertex.getX());
    }

    // Divide the final result by 2 to get the signed area
    area /= 2.0f;

    return area;
}
```

Problem 2:

PolynomialPS1.cpp:



```cpp
#include <iostream>
#include "Polynomial.h"
using namespace std;

Polynomial::Polynomial() : fDegree(0)
{
    for (size_t i = 0; i <= MAX_DEGREE; i++) {
        fCoeffs[i] = 0.0;
    }
}

Polynomial Polynomial::operator*(const Polynomial& aRHS) const {
    Polynomial multiplyResult;
    multiplyResult.fDegree = fDegree + aRHS.fDegree;

    // Multiply polynomials using nested loops
    for (size_t i = 0; i <= fDegree; i++) {
        for (size_t j = 0; j <= aRHS.fDegree; j++) {
            multiplyResult.fCoeffs[i + j] += fCoeffs[i] * aRHS.fCoeffs[j];
        }
    }

    return multiplyResult;
}

bool Polynomial::operator==(const Polynomial& aRHS) const
{
    bool Result = fDegree == aRHS.fDegree;

    // Compare coefficients of polynomials
    for (size_t i = 0; i <= fDegree; i++)
    {
        if (fCoeffs[i] != aRHS.fCoeffs[i])
        {
            Result = false;
        }
    }
    return Result;
}
```

```
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     }
```

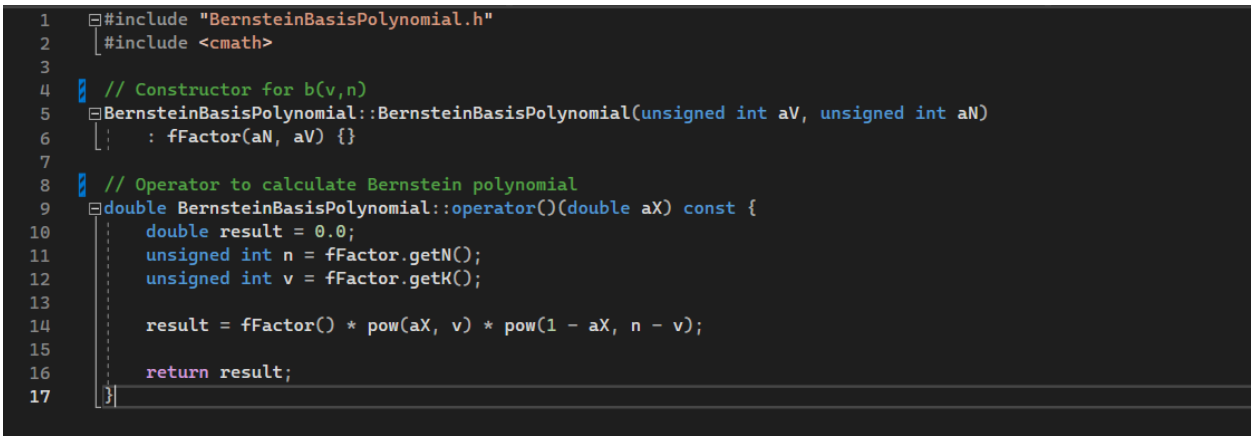
Problem 3:

Combination.cpp:

```cpp
#include "Combination.h"

Combination::Combination(size_t aN, size_t aK)
    : fN(aN), fK(aK)
{
}

size_t Combination::getN() const
{
    return fN;
}

size_t Combination::getK() const
{
    return fK;
}

unsigned long long Combination::operator()() const
{
    unsigned long long result = 1;

    // Calculate the combination using the formula
    // (n choose k) = n! / (k! * (n - k)!)
    for (size_t i = 1; i <= fK; ++i)
    {
        // Multiply the result by (fN - i + 1)
        // to compute the numerator of the combination formula
        result *= fN - i + 1;

        // Divide the result by i
        // to compute the denominator of the combination formula
        result /= i;
    }

    return result;
}
```

Problem 4:

BernsteinBasePolynomial.cpp



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

// Constructor for b(v,n)
BernsteinBasisPolynomial::BernsteinBasisPolynomial(unsigned int aV, unsigned int aN)
    : fFactor(aN, aV) {}

// Operator to calculate Bernstein polynomial
double BernsteinBasisPolynomial::operator()(double aX) const {
    double result = 0.0;
    unsigned int n = fFactor.getN();
    unsigned int v = fFactor.getK();

    result = fFactor() * pow(aX, v) * pow(1 - aX, n - v);

    return result;
}
```