

**Swinburne University (Vietnam)**  
Department of Information Technology

**ASSIGNMENT COVER SHEET**

---

**Subject Code:** COS30008  
**Subject Title:** Data Structures and Patterns  
**Assignment number and title:** 1, Solution Design in C++  
**Due date:** Friday, September 30, 2022, 23:59  
**Lecturer:** Dr. Phuong Anh Nguyen

---

**Your name:** \_\_\_\_\_ **Your student ID:** \_\_\_\_\_

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: \_\_\_\_\_

## Problem Set 1: Solution Design in C++

### Preliminaries

The goal of this problem set is to extend the solution of tutorial 2 and 3. In particular, we wish to add methods to calculate the signed area of a polygon and to determine a polynomial's derivative, its indefinite and definite integral, and its value. In addition, we take a brief look at Bernstein polynomials that form the basis for Bézier curves, parametric curves used in computer graphics and engineering design.

To calculate a polynomial, we need to provide a value for the variable  $x$ . Again, this allows for a straightforward mapping to a for-loop in C++. To compute the  $x^i$ , we need to use the function `pow`, called *raise-to-power*, which is defined in `cmath`. For more details and uses of `pow`, see <http://www.cplusplus.com/reference/cmath/pow/> or Microsoft's documentation at <https://docs.microsoft.com/en-us/cpp/standard-library/cmath?view=msvc-160>.

Please refer to PolynomialMath.pdf available on Canvas to review how one can compute the derivative and integral of a polynomial. An understanding of the mathematical principles is essential for the successful completion of this assignment.

### Conditional Compilation

The test driver provided for this problem set makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

This problem set is comprised of four problems. The test driver (i.e., `main.cpp`) uses `P1`, `P2`, `P3`, and `P4` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective `#define` line. For example, to test the second problem only, enable `#define P2`:

```
// #define P1
#define P2
// #define P3
// #define P4
```

In Visual Studio, the code blocks enclosed in `#ifdef PX ... #endif` are grayed out, if the corresponding test is disabled. The preprocessor definition `#ifdef PX ... #endif` enables conditional compilation. XCode does not use this color coding scheme.

In addition, the test driver defines `#define Automate` and `#ifdef Automate ... #endif`. If you enable `Automate`, then the test driver will simulate the input for problem 2.

## Problem 1

Carl Friedrich Gauss and Carl Gustav Jacob Jacobi invented the trapezoid formula to calculate the area of a trapezoid in the 18<sup>th</sup> century. It forms the basis of the *shoelace algorithm* to determine the area of a simple polygon whose vertices are described by their Cartesian coordinates in the plane. In addition to determining the area of a polygon, the shoelace algorithm also allows us to determine the orientation of a polygon. In particular, if the signed area calculated by the shoelace algorithm is greater than zero, then the vertices are ordered counterclockwise, otherwise the vertices are ordered clockwise. 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.

The algorithm is given by the following formula:

$$A = \frac{1}{2} \left( \begin{vmatrix} x_0 & x_1 \\ y_0 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \cdots + \begin{vmatrix} x_n & x_0 \\ y_n & y_0 \end{vmatrix} \right)$$

where

$$\begin{vmatrix} x_i & x_{i+1} \\ y_i & y_{i+1} \end{vmatrix} = (x_i y_{i+1} - y_i x_{i+1})$$

yields the determinant for the matrix that we obtain by arranging two vectors,  $v_i$  and  $v_{i+1}$ , in column-major matrix form.

Please note that the last determinant connects the last vertex with the first in the polygon.

In order to 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.

```
#pragma once

#include "Vector2D.h"

#define MAX_VERTICES 30

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

public:
    Polygon();

    size_t getNumberOfVertices() const;
    const Vector2D& getVertex( size_t aIndex ) const;

    void readData( std::istream& aIStream );

    float getPerimeter() const;

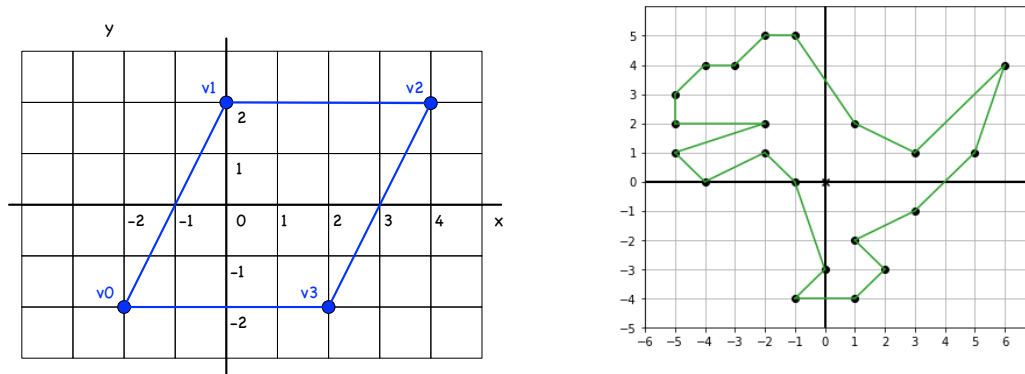
    Polygon scale( float aScalar ) const;

    // Problem Set 1 extension

    float getSignedArea() const;
};
```

Do not edit the provided files. Rather create a new .cpp unit, called `PolygonPS1.cpp`, to implement the new method `getSignedArea()`. Please note that in C++ the implementation of a class does not need to occur in just one compilation unit (i.e., a .cpp file). As long as the

compiler and the linker see all definitions, the build process should succeed, if the program does not contain any errors.



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

There are two sample files that you can use to test your solution: `Parallelogram.txt` and `Data.txt`.

Use as main program the following code:

```
#include "Polygon.h"

void runProblem1( Polygon& aShape )
{
    cout << "Calculating the signed area:" << endl;

    float lArea = aShape.getSignedArea();

    cout << "The area of the polygon is " << fabs( lArea ) << endl;

    if ( lArea > 0.0 )
    {
        cout << "The vertices in the polygon are arranged in counterclockwise order."
              << endl;
    }
    else
    {
        cout << "The vertices in the polygon are arranged in clockwise order." << endl;
    }
}
```

Please note `runProblem1()` is called from the main function, which, in case of problem 1, loads the required data from a text input file (as in tutorial 2). Using `Parallelogram.txt` as input text file, the test code should produce the following result.

```
Microsoft Visual Studio Debug Console
Data read:
Vertex #0: [-2,-2]
Vertex #1: [0,2]
Vertex #2: [4,2]
Vertex #3: [2,-2]
Calculating the signed area:
The area of the polygon is 16
The vertices in the polygon are arranged in clockwise order.
```

## Problem 2

Start with the solution of tutorial 3 (see Canvas). Do not edit the provided files. Rather create a new .cpp unit, called PolynomialPS1.cpp, to implement the new methods.

The extended specification of class Polynomial is shown below. You only need to implement the last four methods. The other features (i.e., constructor and operators) are given as part of the solution for tutorial 3. In the .cpp file for the new methods you need to include `cmath` that contains the definition of `pow` – raise to power.

```
#pragma once

#include <iostream>

#define MAX_POLYNOMIAL 10          // max degree for input
#define MAX_DEGREE MAX_POLYNOMIAL*2+1 // max degree = 10 + 10 + 1 = 21

class Polynomial
{
private:
    size_t fDegree;                // the maximum degree of the polynomial
    double fCoeffs[MAX_DEGREE+1]; // the coefficients (0..10, 0..20)

public:
    // the default constructor (initializes all member variables)
    Polynomial();

    // binary operator* to multiple two polynomials
    // arguments are read-only, signified by const
    // the operator* returns a fresh polynomial with degree i+j
    Polynomial operator*( const Polynomial& aRHS ) const;

    // binary operator== to compare two polynomials
    // arguments are read-only, signified by const
    // the operator== returns true if this polynomial is
    // structurally equivalent to the aRHS polynomial
    bool operator==( const Polynomial& aRHS ) const;

    // input operator for polynomials (highest to lowest)
    friend std::istream& operator>>( std::istream& aIStream,
                                     Polynomial& aObject );

    // output operator for polynomials (highest to lowest)
    friend std::ostream& operator<<( std::ostream& aOStream,
                                     const Polynomial& aObject );

    // Problem Set 1 extension

    // call operator to calculate polynomial for a given x (i.e., aX)
    double operator()( double aX ) const;

    // compute derivative: the derivative is a fresh polynomial with degree
    // fDegree-1, the method does not change the current object
    Polynomial getDerivative() const;

    // compute indefinite integral: the indefinite integral is a fresh
    // polynomial with degree fDegree+1
    // the method does not change the current object
    Polynomial getIndefiniteIntegral() const;

    // calculate definite integral: the method does not change the current
    // object; the method computes the indefinite integral and then
    // calculates it for xlow and xhigh and returns the difference
    double getDefiniteIntegral( double aXLow, double aXHigh ) const;
};
```

Please note the changed value of `MAX_DEGREE`. Since we wish to build integrals, we need to reserve one more slot in the array for the coefficients of a polynomial. Our program supports polynomials up to the 10<sup>th</sup> degree. Multiplying them results in a polynomial up to the 20<sup>th</sup> degree, whose integral is up to the 21<sup>st</sup> degree. Hence, we need 22 entries in the array of coefficients (21+1).

Use as main program the following code:

```
#include <iostream>

#include "Polynomial.h"

using namespace std;

void runProblem2()
{
    Polynomial A;
    cout << "Specify polynomial:" << endl;
    cin >> A;
    cout << "A = " << A << endl;

    double x;
    cout << "Specify value of x:" << endl;
    cin >> x;

    cout << "A(x) = " << A(x) << endl;

    Polynomial B;

    if (B == B.getDerivative())
    {
        cout << "Derivative programmatically sound." << endl;
    }
    else
    {
        cout << "Derivative is broken." << endl;
    }

    if (A == A.getIndefiniteIntegral().getDerivative())
    {
        cout << "Polynomial operations are sound." << endl;
    }
    else
    {
        cout << "Polynomial operations are broken." << endl;
    }

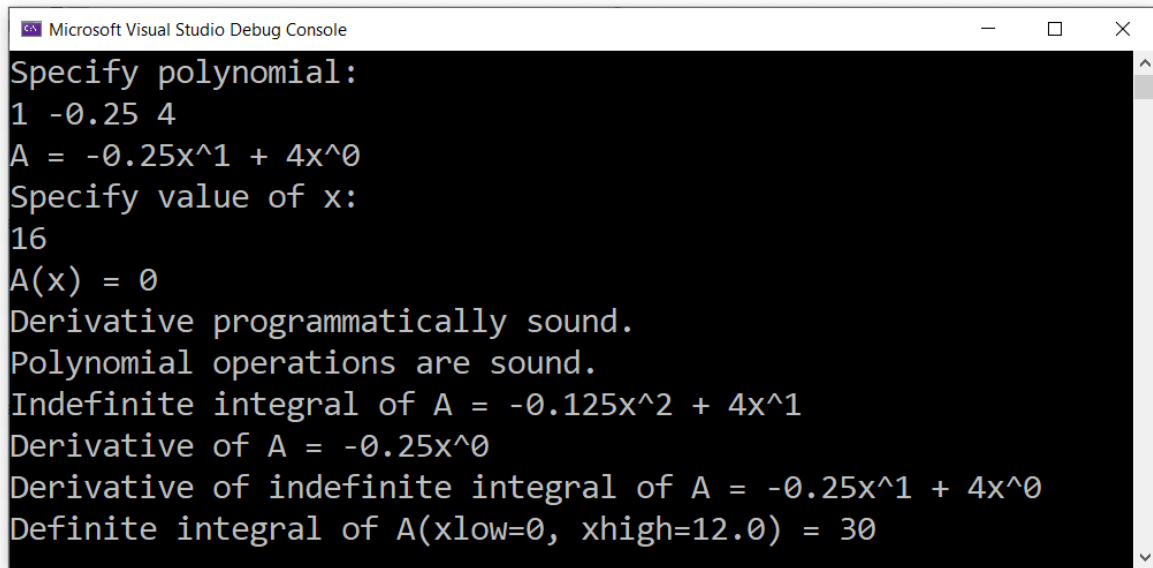
    cout << "Indefinite integral of A = "
        << A.getIndefiniteIntegral() << endl;

    cout << "Derivative of A = "
        << A.getDerivative() << endl;

    cout << "Derivative of indefinite integral of A = "
        << A.getIndefiniteIntegral().getDerivative() << endl;

    cout << "Definite integral of A(xlow=0, xhigh=12.0) = "
        << A.getDefiniteIntegral(0, 12.0) << endl;
}
```

Using  $-0.25x + 4.0$  and  $16$  for the first calculation, the test code should produce the following result.

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the Visual Studio logo and the text "Microsoft Visual Studio Debug Console". The console output is as follows:

```
Specify polynomial:  
1 -0.25 4  
A = -0.25x^1 + 4x^0  
Specify value of x:  
16  
A(x) = 0  
Derivative programmatically sound.  
Polynomial operations are sound.  
Indefinite integral of A = -0.125x^2 + 4x^1  
Derivative of A = -0.25x^0  
Derivative of indefinite integral of A = -0.25x^1 + 4x^0  
Definite integral of A(xlow=0, xhigh=12.0) = 30
```

### Problem 3

In this task, we define a simple data type to represent binomial coefficients

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

A binomial coefficient is a number that denotes number of combinations we can obtain from selection of  $k$  items from a collection with  $n$  elements, such that the order of selection does not matter.

Here, we are interested only in the binomial coefficient

$$\frac{n!}{k!(n-k)!}$$

Whenever  $k \leq n$ , and which is zero when  $k > n$ .

Define a C++ class that implements combination. The specification of class `Combination` is shown below:

```
#pragma once

#include <cstdint>

class Combination
{
private:
    size_t fN;
    size_t fK;

public:
    // constructor for combination n over k with defaults
    Combination( size_t aN = 0, size_t aK = 0 );

    // getters
    size_t getN() const;
    size_t getK() const;

    // call operator to calculate n over k
    // We do not want to evaluate factorials.
    // Rather, we use this method
    //
    //      n      (n-0)   (n-1)      (n - (k - 1))
    //  (   ) =  ----- * ----- * ... * -----
    //      k          1       2          k
    //
    // which maps to a simple for-loop over 64-bit values.
    // https://en.wikipedia.org/wiki/Combination
    unsigned long long operator() () const;
};
```

The class `Combination` has two instance variables representing  $n$  and  $k$ . The constructor initializes those instance variables and the getters provide read-only access to their values.

The call `operator()` returns the value of a combination. We use the method described on Wikipedia: <https://en.wikipedia.org/wiki/Combination> rather than calculating the factorials, which can become very big numbers. You need to use the type `unsigned long long` for calculation in order to work with 64-bit values.



To test your implementation of class `Combination`, we can use the following test driver.

```
#include "Combination.h"

void runProblem3()
{
    // first 10 levels of Pascal's triangle

    cout << "The first ten levels of Pascal's triangle:" << endl;

    for ( size_t n = 0; n < 10; n++ )
    {
        cout << "(n=" << n << ", 0<=k<=" << n << "):";

        int lLead = ((10 - static_cast<int>(n))) * 3;
        for ( size_t k = 0; k <= n; k++ )
        {
            Combination lC( n, k );

            cout << setw( lLead ) << " " << setw(3) << lC();
            lLead = 3;
        }

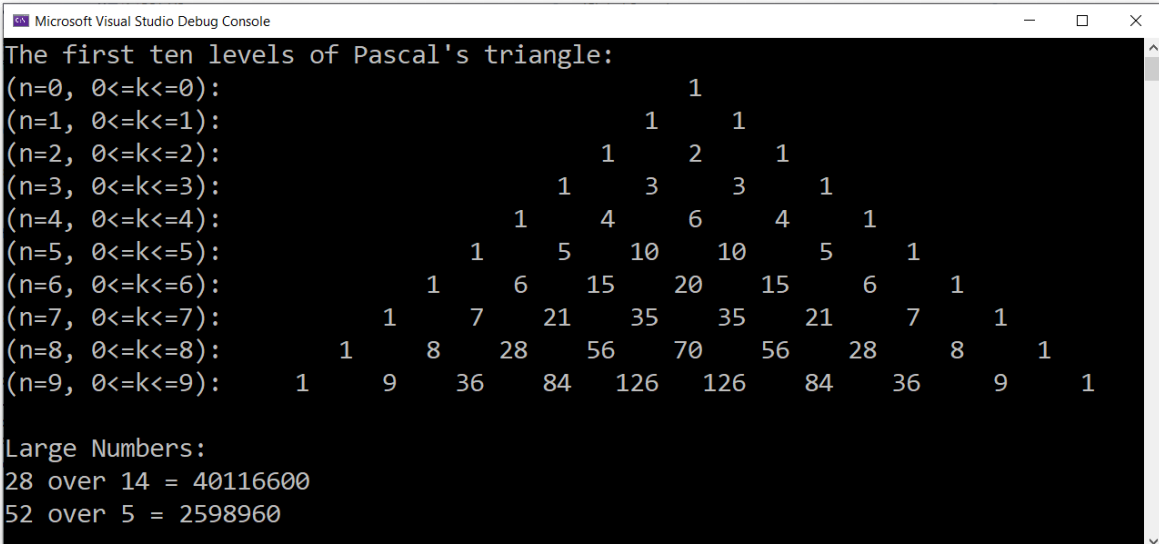
        cout << endl;
    }

    cout << "\nLarge Numbers:" << endl;

    Combination a(28, 14);
    Combination b(52, 5);

    cout << a.getN() << " over " << a.getK() << " = " << a() << endl;
    cout << b.getN() << " over " << b.getK() << " = " << b() << endl;
}
```

The test code should produce the following result.



```
Microsoft Visual Studio Debug Console
The first ten levels of Pascal's triangle:
(n=0, 0<=k<=0):
              1
(n=1, 0<=k<=1):
            1  1
(n=2, 0<=k<=2):
          1  2  1
(n=3, 0<=k<=3):
        1  3  3  1
(n=4, 0<=k<=4):
      1  4  6  4  1
(n=5, 0<=k<=5):
    1  5 10 10  5  1
(n=6, 0<=k<=6):
  1  6 15 20 15  6  1
(n=7, 0<=k<=7):
1  7 21 35 35 21  7  1
(n=8, 0<=k<=8):
1  8 28 56 70 56 28  8  1
(n=9, 0<=k<=9):
1  9 36 84 126 126 84 36  9  1

Large Numbers:
28 over 14 = 40116600
52 over  5 = 2598960
```

## Problem 4

Bézier curves are a fundamental tool in computer graphics and engineering design. They are named after Pierre Bézier who developed them for designing curves for the bodywork of Renault cars. A central building block of Bézier curves is a special type of polynomial – Bernstein basis polynomials (see [https://en.wikipedia.org/wiki/Bernstein\\_polynomial](https://en.wikipedia.org/wiki/Bernstein_polynomial)) – that is restricted to the interval  $[0,1]$ . Bernstein basis polynomials serve as a blending function between control points that are used to interpolate a curve.

In this problem, we only consider Bernstein basis polynomials, which are defined as follows:

$$b_{v,n}(x) = \binom{n}{v} x^v (1-x)^{n-v}, \quad v = 0, \dots, n$$

where  $\binom{n}{v}$  represents a binomial coefficient. For example

$$b_{2,5}(x) = \binom{5}{2} x^2 (1-x)^{5-2} = 10x^2(1-x)^3$$

Every Bernstein basis polynomial  $b_{v,n}(x)$  creates a curve in the interval  $[0,1]$ .

Define a C++ class that implements Bernstein base polynomials. The specification of class `BernsteinBasisPolynomial` is shown below

```
#pragma once

#include "Combination.h"

// https://en.wikipedia.org/wiki/Bernstein_polynomial
class BernsteinBasisPolynomial
{
private:
    Combination fFactor;

public:
    // constructor for b(v,n) with defaults
    BernsteinBasisPolynomial( unsigned int aV = 0, unsigned int aN = 0 );

    // call operator to calculate Bernstein base
    // polynomial for a given x (i.e., aX)
    double operator()( double aX ) const;
};
```

The class `BernsteinBasisPolynomial` has one instance variable to represent the required binomial coefficient. There are two overloaded constructors: a default constructor and a constructor with two arguments. Please observe the order of `aV` and `aN` which is different compared to that in class `Combination` (i.e., reversed).

The call `operator()` evaluates Bernstein basis polynomial object for a given value of `aX`. We assume that `aX` is always in the interval  $[0,1]$ .

To test your implementation of class `BernsteinBasisPolynomial`, we can use the following test driver.

```
#include <iostream>

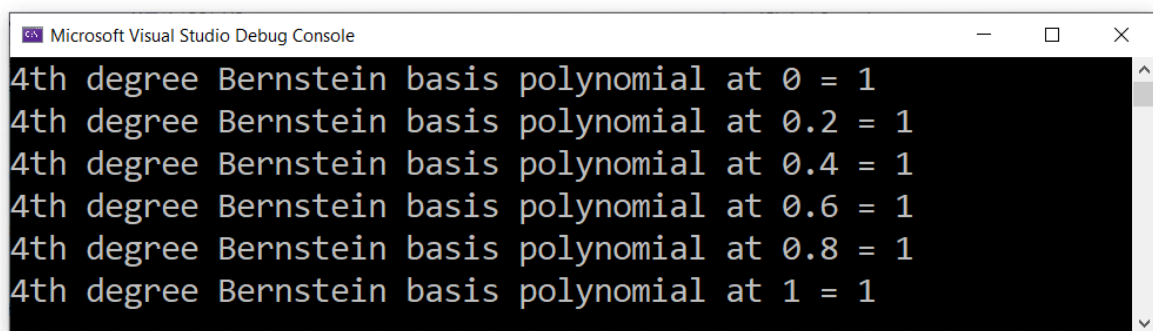
#include "BernsteinBasisPolynomial.h"

void runProblem4()
{
    BernsteinBasisPolynomial bba(0, 4);
    BernsteinBasisPolynomial bbb(1, 4);
    BernsteinBasisPolynomial bbc(2, 4);
    BernsteinBasisPolynomial bbd(3, 4);
    BernsteinBasisPolynomial bbe(4, 4);

    for (double i = 0.0; i < 1.1; i += 0.2)
    {
        cout << "4th degree Bernstein basis polynomial at "
              << i << " = "
              << bba(i) + bbb(i) + bbc(i) + bbd(i) + bbe(i)
              << endl;
    }
}
```

The test code should produce the following result. Please note that the test code interpolates a line made of curve points between  $x$  in  $[0,1]$  that gives  $y = 1$ :

$$\sum_{v=0}^4 b_{v,4}(x) = 1, \quad \forall x. 0 \leq x \leq 1$$



```
Microsoft Visual Studio Debug Console

4th degree Bernstein basis polynomial at 0 = 1
4th degree Bernstein basis polynomial at 0.2 = 1
4th degree Bernstein basis polynomial at 0.4 = 1
4th degree Bernstein basis polynomial at 0.6 = 1
4th degree Bernstein basis polynomial at 0.8 = 1
4th degree Bernstein basis polynomial at 1 = 1
```

The solution of this problem set requires approx. 140-160 lines of low density C++ code.

**Submission deadline: Friday, September 30, 2022, 23:59.**

**Submission procedure:** PDF of printed code for code of `PolygonPS1`, `PolynomialPS1`, `Combination`, and `BernsteinBasePolynomial`. In addition, upload the source code of these files to Canvas, so that we can run your solution when necessary.