COS30008 Semester 1, 2022 Dr. Markus Lumpe

# 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:** Friday, September 30, 2022, 20:59

**Lecturer:** Dr. Markus Lumpe

## Your name:

Tran Quoc Dung

**Your student ID: 103803891**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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:

1

**Problem Set 1**

**Problem 1:**

File: Vector2D.h

// COS30008, 2022

#pragma once

#include <iostream>

class Vector2D

{

private:

float fX;

float fY;

public:

Vector2D(float aX = 1.0f, float aY = 0.0f) : fX(aX), fY(aY) {}

Vector2D(std::istream& aIStream) { aIStream >> \*this; }

float getX() const;

float getY() const;

Vector2D operator+(const Vector2D& aVector) const;

Vector2D operator-(const Vector2D& aVector) const;

Vector2D operator\*(const float aScalar) const;

float dot(const Vector2D& aVector) const;

float cross(const Vector2D& aVector) const;

float length() const;

Vector2D normalize() const;

float direction() const;

Vector2D align(float aAngleInDegrees) const;

friend std::istream& operator>>(std::istream& aIStream, Vector2D& aVector);

friend std::ostream& operator<<(std::ostream& aOStream, const Vector2D& aVector);

};

Vector2D operator\*(const float aScalar, const Vector2D& aVector);

File: Vector2D.cpp

// COS30008, 2022

#define \_USE\_MATH\_DEFINES // must be defined before any #include

#include "Vector2D.h"

#include <cmath>

using namespace std;

float Vector2D::getX() const

{

return fX;

}

float Vector2D::getY() const

{

return fY;

}

Vector2D Vector2D::operator+(const Vector2D& aVector) const

{

return Vector2D(fX + aVector.fX, fY + aVector.fY);

}

Vector2D Vector2D::operator-(const Vector2D& aVector) const

{

return Vector2D(fX - aVector.fX, fY - aVector.fY);

}

Vector2D Vector2D::operator\*(const float aScalar) const

{

return Vector2D(fX \* aScalar, fY \* aScalar);

}

float Vector2D::dot(const Vector2D& aVector) const

{

return fX \* aVector.fX + fY \* aVector.fY;

}

float Vector2D::cross(const Vector2D& aVector) const

{

return fX \* aVector.fY - fY \* aVector.fX;

}

float Vector2D::length() const

{

float val = sqrt(fX \* fX + fY \* fY);

return round(val \* 100.0f) / 100.0f;

}

Vector2D Vector2D::normalize() const

{

return \*this \* (1.0f / length());

}

float Vector2D::direction() const

{

float val = atan2(fY, fX) \* 180.0f / static\_cast<float>(M\_PI);

return round(val \* 100.0f) / 100.0f;

}

Vector2D Vector2D::align(float aAngleInDegrees) const

{

float lRadians = aAngleInDegrees \* static\_cast<float>(M\_PI) / 180.0f;

return length() \* Vector2D(cos(lRadians), sin(lRadians));

}

istream& operator>>(istream& aIStream, Vector2D& aVector)

{

return aIStream >> aVector.fX >> aVector.fY;

}

ostream& operator<<(ostream& aOStream, const Vector2D& aVector)

{

return aOStream << "[" << round(aVector.fX) << "," << round(aVector.fY) << "]";

}

Vector2D operator\*(const float aScalar, const Vector2D& aVector)

{

return aVector \* aScalar;

}

File: Polygon.h

#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;

};

File: PolygonPS1.cpp

// COS30008, Tutorial 2, 2022

#include "Polygon.h"

#include <stdexcept>

using namespace std;

Polygon::Polygon() :

fNumberOfVertices(0)

{}

size\_t Polygon::getNumberOfVertices() const

{

return fNumberOfVertices;

}

const Vector2D& Polygon::getVertex(size\_t aIndex) const

{

if (aIndex < fNumberOfVertices)

{

return fVertices[aIndex];

}

throw out\_of\_range("Illegal index value.");

}

void Polygon::readData(istream& aIStream)

{

// read input file containing 2D vector data

// if no data can be read, then exit loop

// lInput >> lVectors[lIndex] evaluates to false on EOF

while (aIStream >> fVertices[fNumberOfVertices])

{

fNumberOfVertices++;

}

}

float Polygon::getPerimeter() const

{

float Result = 0.0f;

// There have to be at least three vertices

if (fNumberOfVertices > 2)

{

// solution without modulus and explicit temporary variables

for (size\_t i = 1; i < fNumberOfVertices; i++)

{

Result += (fVertices[i] - fVertices[i - 1]).length();

}

Result += (fVertices[0] - fVertices[fNumberOfVertices - 1]).length();

}

return Result;

}

Polygon Polygon::scale(float aScalar) const

{

Polygon Result = \*this;

for (size\_t i = 0; i < fNumberOfVertices; i++)

{

Result.fVertices[i] = fVertices[i] \* aScalar;

}

return Result;

}

float Polygon::getSignedArea() const

{

float Area = 0.0f;

for (size\_t i = 0; i < getNumberOfVertices() - 1; i++)

{

Area += (fVertices[i].getX() \* fVertices[i + 1].getY());

Area -= (fVertices[i + 1].getX() \* fVertices[i].getY());

}

Area += (fVertices[getNumberOfVertices() - 1].getX() \* fVertices[0].getY());

Area -= (fVertices[getNumberOfVertices() - 1].getY() \* fVertices[0].getX());

Area /= 2;

return Area;

}

**Problem 2:**

File: Polynomial.h

#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;

};

File: PolynomialPS1.cpp

#include "Polynomial.h"

#include <math.h>

using namespace std;

Polynomial::Polynomial() :

fDegree(0)

{

for (size\_t i = 0; i <= MAX\_DEGREE; i++)

{

fCoeffs[i] = 0.0;

}

}

bool Polynomial::operator==(const Polynomial& aRHS) const

{

bool Result = fDegree == aRHS.fDegree;

for (size\_t i = 0; Result && i <= fDegree; i++)

{

if (fCoeffs[i] != aRHS.fCoeffs[i])

{

Result = false;

}

}

return Result;

}

Polynomial Polynomial::operator\*(const Polynomial& aRight) const

{

// C = A \* B

Polynomial Result;

Result.fDegree = fDegree + aRight.fDegree;

for (size\_t i = 0; i <= fDegree; i++)

{

for (size\_t j = 0; j <= aRight.fDegree; j++)

{

Result.fCoeffs[i + j] += fCoeffs[i] \* aRight.fCoeffs[j];

}

}

return Result;

}

ostream& operator<<(ostream& aOStream, const Polynomial& aObject)

{

bool lMustPrintPlus = false;

for (int i = static\_cast<int>(aObject.fDegree); i >= 0; i--)

{

if (aObject.fCoeffs[i] != 0.0)

{

if (lMustPrintPlus)

{

aOStream << " + ";

}

else

{

lMustPrintPlus = true;

}

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

}

}

return aOStream;

}

istream& operator>>(istream& aIStream, Polynomial& aObject)

{

// read degree

size\_t lDegree;

aIStream >> lDegree;

aObject.fDegree = lDegree <= MAX\_POLYNOMIAL ? lDegree : MAX\_POLYNOMIAL;

// read coefficients (assume sound input)

for (int i = static\_cast<int>(aObject.fDegree); i >= 0; i--)

{

aIStream >> aObject.fCoeffs[i];

}

return aIStream;

}

double Polynomial::operator()(double aX) const

{

double value = 0;

for (int i = 0; i <= fDegree; i++)

{

value += (pow(aX, i) \* fCoeffs[i]);

}

return value;

}

Polynomial Polynomial::getDerivative() const

{

Polynomial D;

if (fDegree > 0)

{

D.fDegree = fDegree - 1;

}

for (size\_t i = 0; i <= D.fDegree; i++)

{

D.fCoeffs[i] = fCoeffs[i + 1] \* (i + 1);

}

return D;

}

Polynomial Polynomial::getIndefiniteIntegral() const

{

Polynomial I;

I.fDegree = fDegree + 1;

for (size\_t i = 1; i <= I.fDegree; i++)

{

I.fCoeffs[i] = fCoeffs[i - 1] / i;

}

return I;

}

double Polynomial::getDefiniteIntegral(double axLow, double axHigh) const {

double result = 0;

Polynomial I = getIndefiniteIntegral();

result = I(axHigh) - I(axLow);

return result;

}

**Problem 3:**

File: Combination.cpp

#pragma once

#include <iostream>

#include <cstddef>

using namespace std;

class Combination

{

private:

size\_t fN;

size\_t fK;

public:

Combination(size\_t n, size\_t k) {

fN = n;

fK = k;

}

size\_t getN() { return fN; }

size\_t getK() { return fK; }

// 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.

public:

unsigned long long Operator1() {

long long result = 1;

for (int i = 1; i <= fK; i++) {

result = result \* (fN - i + 1) / i;

}

return result;

}

};

// write "Combination\_name".setN for fN value

// write "Combination\_name".setK for fK value

**Problem 4:**

File: BernsteinBasicPolynomial.cpp

#pragma once

#include <iostream>

#include <cmath>

#include "Combination.cpp"

using namespace std;

class BernsteinBasisPolynomial

{

private:

unsigned int fN;

unsigned int fV;

public:

// constructor for b(v,n) with defaults

BernsteinBasisPolynomial(unsigned int aN, unsigned int aV) {

fN = aN;

fV = aV;

}

// call operator to calculate Berstein base

// polynomial for a given x (i.e., aX)

double Operator2(double x) const {

double result;

Combination factor(fN, fV);

result = factor.Operator1() \* pow(x, fV) \* pow((1 - x), (fN - fV));

return result;

}

};