Bachelor of Computer Science
MIDTERM

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: Midterm, Convex Hull
Due date: Check Canvas

Your name: Gia Huy Nguyen **Your student ID: 103441107**

Marker's comments:

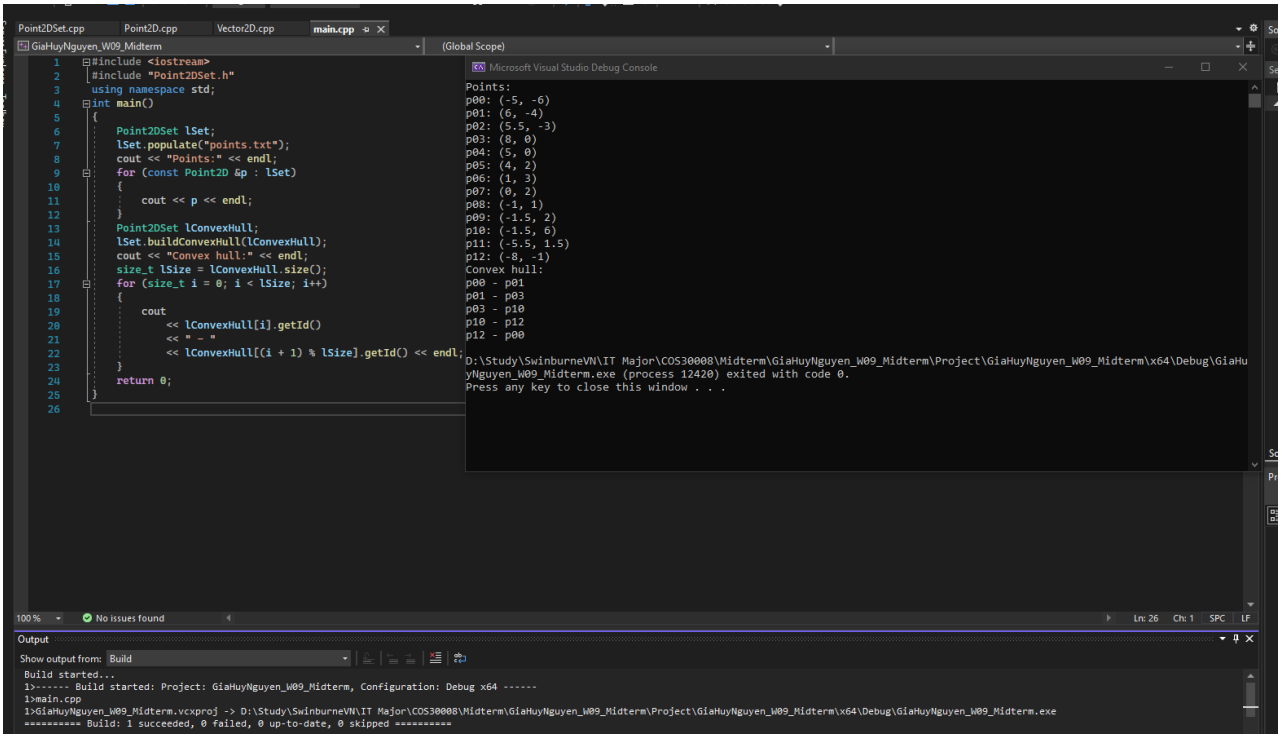
Problem	Marks	Obtained
1	20	
2	30	
3	50 (buildConvexHull:20)	
Total	100	

Detailed comments:

Test Driver (main.ccp)

```
#include <iostream>
#include "Point2DSet.h"
using namespace std;
int main()
{
    Point2DSet lSet;
    lSet.populate("points.txt");
    cout << "Points:" << endl;
    for (const Point2D &p : lSet)
    {
        cout << p << endl;
    }
    Point2DSet lConvexHull;
    lSet.buildConvexHull(lConvexHull);
    cout << "Convex hull:" << endl;
    size_t lSize = lConvexHull.size();
    for (size_t i = 0; i < lSize; i++)
    {
        cout
            << lConvexHull[i].getId()
            << " - "
            << lConvexHull[(i + 1) % lSize].getId() << endl;
    }
    return 0;
}
```

Output (for points.txt):



Problem 1 (Vector2D.cpp)

```cpp
#include "Vector2D.h"
#include <iostream>
#include <string>
#include <math.h>

using namespace std;
// 2D vector constructor; default is unit vector î
Vector2D::Vector2D(double aX, double aY)
{
    fX = aX;
    fY = aY;
}
// getters & setters for x and y coordinates
void Vector2D::setX(double aX)
{
    fX = aX;
}
double Vector2D::getX() const
{
    return fX;
}
void Vector2D::setY(double aY)
{
    fY = aY;
}
double Vector2D::getY() const
{
    return fY;
}
// 2D vector addition: this + aRHS; returns a fresh 2D vector
Vector2D Vector2D::operator+(const Vector2D &aRHS) const
{
    return {fX + aRHS.fX, fY + aRHS.fY};
}
// 2D vector subtraction: this - aRHS; returns a fresh 2D vector
Vector2D Vector2D::operator-(const Vector2D &aRHS) const
{
    return {fX - aRHS.fX, fY - aRHS.fY};
}
// Length (or magnitude) of a 2D vector
double Vector2D::magnitude() const
{
    return sqrt(fX * fX + fY * fY);
}
// Direction (angle) of 2D vector
// The angle is the tangent of y divided by x (hint: atan2)
double Vector2D::direction() const
{
    return atan2(fY, fX);
}
// Inner product (scalar product) of two 2D vectors
// Does not require angle between vectors
double Vector2D::dot(const Vector2D &aRHS) const
{
    return fX * aRHS.fX + fY * aRHS.fY;
}
// In 2D, the cross product of two 2D vectors is
// the determinate of the matrix
//
// | x1 x2 |
// det | | = x1*y2 - x2*y1
// | y1 y2 |
```

```
//
double Vector2D::cross(const Vector2D &aRHS) const
{
    return fX * aRHS.fY - fY * aRHS.fX;
}
// Angle between two 2D vectors
// The function must properly handle null vectors = [0,0]
// and return an angle consistent with the dot product.
double Vector2D::angleBetween(const Vector2D &aRHS) const
{
    double dot = fX * aRHS.fX + fY * aRHS.fY;
    double mag = magnitude() * aRHS.magnitude();
    if (mag == 0.0)
    {
        return 0.0;
    }
    return acos(dot / mag);
}
// I/O for 2D vectors
std::ostream &operator<<(std::ostream &aOutStream, const Vector2D &aObject)
{
    aOutStream << "[" << aObject.fX << ", " << aObject.fY << "]";
    return aOutStream;
}
std::istream &operator>>(std::istream &aInStream, Vector2D &aObject)
{
    aInStream >> aObject.fX >> aObject.fY;
    return aInStream;
}
```

Problem 2 (Point2D.cpp)

```cpp
#include "Point2D.h"
#include <iostream>
#include <string>
#include <math.h>
#include <cmath>
#define M_PI 3.14159265358979323846

using namespace std;
static const Point2D gCoordinateOrigin = Point2D("origin", 0, 0);
// Constructor
Point2D::Point2D()
{
    fId = "";
    fPosition = Vector2D(0, 0);
    fOrigin = &gCoordinateOrigin;
}
Point2D::Point2D(const std::string &aId, double aX, double aY)
{
    fId = aId;
    fPosition = Vector2D(aX, aY);
    fOrigin = &gCoordinateOrigin;
}
Point2D::Point2D(std::istream &aIStream)
{
    aIStream >> fId >> fPosition;
    fOrigin = &gCoordinateOrigin;
}
// getters & setters
const std::string &Point2D::getId() const
{
    return fId;
}
void Point2D::setX(const double &aX)
{
    fPosition.setX(aX);
}
const double Point2D::getX() const
{
    return fPosition.getX();
}
void Point2D::setY(const double &aY)
{
    fPosition.setY(aY);
}
const double Point2D::getY() const
{
    return fPosition.getY();
}
void Point2D::setOrigin(const Point2D &aPoint)
{
    fOrigin = &aPoint;
}
const Point2D &Point2D::getOrigin() const
{
    return *fOrigin;
}
// Direction (angle) of point w.r.t. aOther
double Point2D::directionTo(const Point2D &aOther) const
{
    Vector2D v = aOther.fPosition - fPosition;
    return v.direction();
}
```

```
// Length ( or magnitude) of point w.r.t. aOther
double Point2D::magnitudeTo(const Point2D &aOther) const
{
    Vector2D v = aOther.fPosition - fPosition;
    return v.magnitude();
}
// 2D vector this - aRHS
Vector2D Point2D::operator-(const Point2D &aRHS) const
{
    return fPosition - aRHS.fPosition;
}
// Direction (angle) of point w. r.t. origin
double Point2D::direction() const
{
    return fOrigin->directionTo(*this);
}
// Length (or magnitude) of point w.r.t. origin
double Point2D::magnitude() const
{
    return fOrigin->magnitudeTo(*this);
}
// Are this point and aOther on the same line segment?
bool Point2D::isCollinear(const Point2D &aOther) const
{
    double d = directionTo(aOther);
    return (abs(d) <= 1e-4 || abs(d - M_PI) <= 1e-4);
}
// Does line segment this-aP2 make a right turn at this point?
bool Point2D::isClockwise(const Point2D &aP0, const Point2D &aP2) const
{
    double val = (fPosition.getY() - aP0.fPosition.getY()) * (aP2.fPosition.getX() -
fPosition.getX()) -
                 (fPosition.getX() - aP0.fPosition.getX()) * (aP2.fPosition.getY() -
fPosition.getY());
    if (val < 0)
    {
        return false;
    }
    return true;
}
//// Is this' y-coordinate less than aRHS's y-coordinate?
// If there is a tie, this' x-coordinate less than aRHS's x-coordinate?
bool Point2D::operator<(const Point2D &aRHS) const
{
    if (fPosition.getY() < aRHS.fPosition.getY())
    {
        return true;
    }
    else if (fPosition.getY() == aRHS.fPosition.getY())
    {
        if (fPosition.getX() < aRHS.fPosition.getX())
        {
            return true;
        }
    }
    return false;
}
// I/O for 2D points
std::ostream &operator<<(std::ostream &aOStream, const Point2D &aObject)
{
    aOStream << aObject.fId << ": (" << aObject.fPosition.getX() << ", "
             << aObject.fPosition.getY() << ")";
    return aOStream;
}
std::istream &operator>>(std::istream &aIStream, Point2D &aObject)
```

```
{
    aIStream >> aObject.fId >> aObject.fPosition;
    return aIStream;
}
```

Problem 3 (Point2DSet.cpp)

```cpp
#include "Point2DSet.h"
#include <vector>
#include <iostream>
#include <algorithm>
#include <fstream>

using namespace std;
// Add a 2D point to set
void Point2DSet::add(const Point2D& aPoint)
{
    fPoints.push_back(aPoint);
}
void Point2DSet::add(Point2D&& aPoint)
{
    fPoints.push_back(aPoint);
}
// Remove the last point in the set
void Point2DSet::removeLast()
{
    fPoints.pop_back();
}
// Tests aPoint, returns true if aPoint makes no left turn
bool Point2DSet::doesNotTurnLeft(const Point2D& aPoint) const
{
    return fPoints[size() - 1].isClockwise(fPoints[size() - 2], aPoint);
}
// Load 2D points from file
void Point2DSet::populate(const std::string& aFileName)
{
    ifstream inFile(aFileName);
    if (!inFile)
    {
        cout << "Error opening file " << aFileName << endl;
        return;
    }
    Point2D point;
    // number of points in file
    int n;
    inFile >> n;
    for (int i = 0; i < n; i++)
    {
        inFile >> point;
        add(point);
    }
    inFile.close();
}

bool orderByCoordinates(const Point2D& aLeft, const Point2D& aRight)
{
    return aLeft < aRight;
}
bool orderByPolarAngle(const Point2D& aLHS, const Point2D& aRHS)
{
    if (aLHS.isCollinear(aRHS)) {
        return aLHS.magnitude() < aRHS.magnitude() - 1e-4;
    }
    return aLHS.direction() < aRHS.direction() - 1e-4;
}

// Sort set using stable_sort on vectors
void Point2DSet::sort(Comparator aComparator)
{
```

```
    stable_sort(fPoints.begin(), fPoints.end(), aComparator);
}
// Returns numner of elements in set
size_t Point2DSet::size() const
{
    return fPoints.size();
}
// Clears set
void Point2DSet::clear()
{
    fPoints.clear();
}
// Run Graham's scan using out parameter aConvexHull
void Point2DSet::buildConvexHull(Point2DSet& aConvexHull)
{
    aConvexHull.clear();

    sort(orderByCoordinates);
    for (auto& point : fPoints)
    {
        point.setOrigin(fPoints[0]);
    }
    sort(orderByPolarAngle);
    aConvexHull.add(fPoints[0]);
    aConvexHull.add(fPoints[1]);
    aConvexHull.add(fPoints[2]);
    for (size_t i = 3; i < size(); i++)
    {
        while (aConvexHull.size() >= 2 && aConvexHull.doesNotTurnLeft(fPoints[i]))
        {
            aConvexHull.removeLast();
        }
        aConvexHull.add(fPoints[i]);
    }
}
// Indexer for set
const Point2D& Point2DSet::operator[](size_t aIndex) const
{
    return fPoints[aIndex];
}
// Iterator methods
Point2DSet::Iterator Point2DSet::begin() const
{
    return fPoints.begin();
}
Point2DSet::Iterator Point2DSet::end() const
{
    return fPoints.end();
}
```