Farhan Zaman 9/14/18
CS21200 10:00AM

HW 1 Report

The point class creates objects that have 3 coordinates to describe them. I have also created both member and non-member functions to performs actions based of problems that use these points to solve certain things. We have maybe each coordinate be of the double data type since it is very common to have locations of points what are in decimal form.
For question 7 I wrote:

```
void Point::print()
{
	cout<<"("<<x<<","<<y<<","<<z<<")"<<endl;
}
```

This print function prints the coordinates of a point in the format of (x,y,z). I just output each individual parenthesis, and comma and coordinate to get it in that format. There are no preconditions, the post condition is that the coordinate is printed on the screen. There is no worst case time complexity.

For question 8 I wrote:

```
double Point::distance()
{
	double answer;
	answer=sqrt(x*x+y*y+z*z);
	return answer;
}
```

This distance function returns the distance between a point and the origin. The distance between 2 points in the 3-D world is sqrt((x-x0)^2+(y-yo)^2+(z-z0)^2). Since one of the points is 0,0,0, we can plug that in for x0,y0,z0 and simplify. There is no precondition, and the post condition is the distance between both points is returned. There is no worst case time complexity.

```
double Point::distance(Point pt2)
{
	double newx=(x-pt2.getX())*(x-pt2.getX());
	double newy=(y-pt2.getY())*(y-pt2.getY());
	double newz=(z-pt2.getZ())*(z-pt2.getZ());
	return sqrt(newx+newy+newz);
}
```

For question 9 I wrote:

```
bool Point::line(Point pt2)
{
	double scalar;
	if(x!=0 && pt2.x!=0)
	{
		scalar = pt2.x/x;
		return y*scalar == pt2.y && z*scalar==pt2.z;
	}
	else if(y!=0 && pt2.y!=0)
	{
		scalar = pt2.y/y;
		return x*scalar == pt2.x && z*scalar==pt2.z;
	}
	else if(z!=0 && pt2.z!=0)
	{
		scalar = pt2.z/z;
		return y*scalar == pt2.y && x*scalar==pt2.x;
	}

	return false;

}
```

The function line returns true if it is on the line created by a point and the origin. It returns false if there isn't a line created or it isn't on the line. For 2 points in the 3-D world to be on the same line, they have to be scalar multiples of the vectors created with the origin. If one directional vector is <1,2,3> and we want to check if <2,4,6> is on the line, then we check that <2,4,6> is a scalar multiple of the other by dividing each respective components. 2/1=2, 4/2=2, 6/3=2, since the scalar is 2 for all the components, <2,4,6> is on the line. The precondition of this function is that the point and line make a line and aren't the same point. The post condition is a truth value on whether or not pt2 is on the line. The worst time complexity is if it isn't on the line and isn't 0,0,0.

For question 10 I wrote:

```
Point Point::cross(Point pt2)
```

```
{
	Point pt3;
	double inX, inY,inZ;
	inX=(y*pt2.getZ())-(z*pt2.getY());
	inY=-1*((x*pt2.getZ())-( z*pt2.getX()));
	inZ=(x*pt2.getY())-(y*pt2.getX());
	pt3.setXYZ(inX,inY,inZ);
	return pt3;
}
```

The function cross returns the cross product of 2 points. It does this by creating a 3x3 matrix where the first row is i,j,k, the 2nd and 3rd row are the 2 points we are crossing.
[i j k]
[x y x]
[x0 y0 z0]
To find the i coordinate we cover the column and row including i and use the remaining 2x2 matrix which would be
[y x]
[y0 z0]
Here we do (y*z0)-(y0*x). We repeat the process for the j and k coordinate, but the final step is to figure out the coordinates of i,j,k and find out if its even or not. For i it was 1+1=2 which is even so the answer is positive, j was 1+2=3 which is odd so the answer is negative and so on and so forth. There is no precondition. The post condition is a new point which is the cross product is returned. There is no worst case time complexity.

For question 11 I wrote:

```
Point Point::operator +(Point pt2)
{
	Point pt3;
	double inX, inY,inZ;
	inX=x+pt2.getX();
	inY=y+pt2.getY();
	inZ=z+pt2.getZ();
	pt3.setXYZ(inX,inY,inZ);
	return pt3;
}
```

```
Point Point::operator -(Point pt2)
{
	Point pt3;
	double inX, inY,inZ;
	inX=x-pt2.getX();
	inY=y-pt2.getY();
	inZ=z-pt2.getZ();s.
	pt3.setXYZ(inX,inY,inZ);
	return pt3;
}
```

For these functions I overloaded the + and - operators which only worked on int,double,string before to work for the Point class now. If I add 2 points, I want each component to be added respectively. So if we are adding p1 and p2, the x components of both points are added, the same with y and z, and then return a point that has the new component. The same goes for the - operator, except now the order matters so we do p1-p2 where its p1.x-p2.x and then the same as before applies.The worst case time complexity is just big O(1).

For question 12 I wrote:

```
istream& operator >> (istream& input, Point& pt)
{
	input>>pt.x>>pt.y>>pt.z;
	return input;
}

ostream& operator << (ostream& output,const Point& pt)
{
	output<<"("<<pt.x<<","<<pt.y<<","<<pt.z<<")"<<endl;
	return output;
}

bool Point::operator == (Point pt2)
{
	if(x==pt2.x && y==pt2.y && z==pt2.z)
	{
		return true;
	}
	return false;
}
```

This function overloads the >> and << operators so I can use it with points. << and >> typically work for string, int float, double, char, so we overload it such that when 3 numbers are inputted by the user, it is set to the variable, and when we << the variable it is printed ot the screen in the format of (x,y,z). The preconditions of the functions is that we are working with a Point that is called by reference but is const only for the << operator. The post condition for >> is we return a point with coordinates of what the user inputted, and the post condition for << is we print out the point in (x,y,z) format. The worst time complexity doesn't exist.
For question 13 I wrote:

```
bool plane(Point Ar[], Point pt)
{
	Point u,v;
	u=Ar[1]-Ar[0];
	v=Ar[2]-Ar[0];
	Point crossproduct;
	crossproduct=u.cross(v);

if(crossproduct.getX()*(pt.getX()-Ar[0].getX())+crossproduct.getY()*(pt.getY()-Ar[0].
getY())+crossproduct.getZ()*(pt.getZ()-Ar[0].getZ())==0)
	{
		return true;
	}
	return false;
}
```

This plane function returns true if the target point is on the plane created by 3 points. It does this by creating a vector normal to the plane by doing a cross product, and then checking if the target point's cross product with the normal vector is 0, which means it in on the plane, if it isn't 0 then it isn't on the plane. The precondition for this is that there is a plane and the points are unique. The post condition is a truth value is returned if the target point is on the plane or not. The worst case time complexity would be if the point isn't on the plane.

For question 14 I wrote:

```
bool square(Point Ar[], int size)
{
	for(int i=0; i<size; i++)
	{
		Point pt1=Ar[i];
		for(int j=i+1; j<size; j++)
		{
```

```
                        Point pt2=Ar[j];
                        for(int k=j+1; k<size; k++)
                        {
                                Point pt3=Ar[k];
                                for(int l=k+1; l<size; l++)
                                {
                                        Point pt4=Ar[l];

                                        double AB,AC,AD,BC,BD,CD;
                                        AB=pt1.distance(pt2);
                                        AC=pt1.distance(pt3);
                                        AD=pt1.distance(pt4);
                                        BC=pt2.distance(pt3);
                                        BD=pt2.distance(pt4);
                                        CD=pt3.distance(pt4);



                                        if(AB==AC && BD==CD && AB==BD && (sqrt(2)*AB)==AD &&
(sqrt(2)*AB)==BC)
                                        {
                                                return true;
                                        }
                                        if(AB==AD && BC==CD && AB==CD && (sqrt(2)*AB)==AC &&
(sqrt(2)*AB)==BD)
                                        {
                                                return true;
                                        }
                                        if(AC==AD && BC==BD && AC==BC && (sqrt(2)*AC)==AB &&
(sqrt(2)*AC)==CD)
                                        {
                                                return true;
                                        }

                                }
                        }
                }
        }
        return false;
}
```

This square function returns a truth value on whether or not any combination of 4 points from the array creates a square. It does this by first going through every combo of 4 unique points with 4 for loops. Every point is labeled as A,B,C,D, and between 4 points there are 6 distances to be found. For a square to be a square, it must have 4 equal sides, and the diagonals should be sqrt(2)

* the length of the side. This would specifically account for the angle to be 45, and then 2 of these give us 90 degree angles. There are only 3 ways to orient a corner with 4 points, so I wrote 3 if statements to account for those 3 scenarios and if all 3 scenarios don't work, check every 4 point combo, and if all fails then it returns false. The precondition is that the size of the array is <= the capacity of it and that all points are unique. The post condition is a truth value on whether or not any combo of 4 points made a square. The worst case time complexity is that it doesn't form a square at all.

For question 15 I wrote:

```
Point centroid(Point Ar[], int size)
{
	Point centroid;
	double subx=0,suby=0,subz=0;
	for(int i=0; i<size; i++)
	{
		subx+=Ar[i].getX();
	}
	for(int i=0; i<size; i++)
	{
		suby+=Ar[i].getY();
	}
	for(int i=0; i<size; i++)
	{
		subz+=Ar[i].getZ();
	}
	centroid.setXYZ(subx/size,suby/size,subz/size);
	return centroid;
}
```

This function finds the centroid of any number of points. It does this by average all the components values. So the x coordinate for the centroid is the average of x coordinates from every point and the same for y and z. The precondition is that the size of the array is <= its capacity. The postcondition is a point is return that is the center of all the points. The worst case time complexity is the full capacity of the array is filled with points.

The class itself was easy enough to do and understand, but some questions seemed extremely hard to code like the square one, when in real life it would be much easier to see the answer. I didn't entirely understand the usage of the const for the get functions or even the output operator. For the square function initially my for loops were i=0,j=1,k=2,l=3, but I kept getting the wrong answers, and it only worked when i did j=i+1, k=j+1, and l=k+1.

Here is the code for my header file:

```
#ifndef __ZAMAN__FARHAN__HW1_H__
#define __ZAMAN__FARHAN__HW1_H__

#include <iostream>
#include <cassert>
#include <cmath>
#include <cstdio>

using namespace std;

class Point
{
	private:
		double x;
		double y;
		double z;
	public:
		Point();
		Point(double inX, double inY, double inZ=0);
		Point(const Point& input2);
		void setX(double inX);
		void setY(double inY);
		void setZ(double inZ);
		void setXY(double inX, double inY);
		void setXYZ(double inX, double inY, double inZ);
		double getX() const;
		double getY() const;
		double getZ() const;
		void print();
		double distance();
		double distance(Point pt2);
		bool line(Point pt2);
		Point cross(Point pt2);
		Point operator + (Point pt2);
		Point operator - (Point pt2);
		friend istream& operator >> (istream& input, Point& pt);
		friend ostream& operator << (ostream& output, const Point& pt);
};

	bool plane(Point Ar[], Point pt);
	bool square(Point Ar[], int size);
	Point centroid(Point Ar[], int size);
```

```
#endif
```