```
/*
*************************************************************************
*************************************************************************
*******************
Misc. Functions created by Nathan Christian Copyright (c) 2017
*************************************************************************
*************************************************************************
*******************
*/
#include <math.h>
#define _USE_MATH_DEFINES
#include "shape.h"
#define PI M_PI
#include <assert.h>

void print_help(void)
{
      printf("\n\nThis program rotates a shape by a given number of
degrees and\n");
      printf("prints out a fig or .txt file.\n\n");
      printf("Use one or more of the entries below:\n\n");
      printf("\t-H\t ---> help\n");
      printf("\t-V\t ---> prints the current version\n");
      printf("\t-A xx\t ---> Turns the shape by xx degrees. The default
angle is 90 degrees.\n");
      printf("\t-L fig \t --->(REQUIRED) tells the compiler to print a fig
or shape output\n");
      printf("\t-O file\t ---> (REQUIRED) tells the compiler to open a
specific file.\n\n");
      printf("\t Sample files available are 'casette.txt', 'aircraft1.txt'
and aircaft2.txt\n\n");
      printf("\n An example of a valid entry is 'lab5 -o casette.txt -l
fig -a 45'\n");
      exit(1);
}

void high_low(Point *low, Point *high, Point *t_low, Point *t_high)
{
      if (low->x>t_low->x)
                  low->x=t_low->x;
      if(low->y>t_low->y)
                  low->y=t_low->y;
      if(high->x<t_high->x)
                  high->x=t_high->x;
      if(high->y<t_high->y)
                  high->y=t_high->y;
}
/*
*************************************************************************
*************************************************************************
*******************
Circle Functions created by Nathan Christian
```

```
**************************************************************************
**************************************************************************
******************
*/
void circle_boundary(Circle *circle, Point *lower, Point *upper)
{
      lower->x=(circle->center.x)-circle->radius;
      lower->y=(circle->center.y)-circle->radius;
      upper->x=(circle->center.x)+circle->radius;
      upper->y=(circle->center.y)+circle->radius;
}

int circle_move(Circle *circle, double theta, double distance2, struct
Point delta)
{
      double total_angle, angle;
      circle->center.x=(circle->center.x)-delta.x;
      circle->center.y=(circle->center.y)-delta.y;
                                    //Shifts center of shapes to origin for
easy computations

      if (circle->center.x>0 && circle->center.y>0)

      total_angle=theta+atan(circle->center.y/circle->center.x);
                              //Find new angle If initial angle begins in
quadrant 1

      if (circle ->center.x<0 && circle->center.y>0)
                  total_angle=PI+theta-
(atan(fabs(circle->center.y/circle->center.x)));
      //Find new angle if initial angle begins in quadrant 2

      if (circle ->center.x<0 && circle->center.y<0)

      total_angle=PI+theta+(atan(circle->center.y/circle->center.x));
                              //Find new angle if initial angle begins in
quadrant 3

      if (circle->center.x>0 && circle->center.y<0)
                  total_angle=(2*PI)+theta-
(atan(fabs(circle->center.y/circle->center.x)));              //Find new
angle if initial angle begins in quadrant 4

      if (total_angle>(2.0*PI))
                              total_angle =total_angle-(2.0*PI);
                                                    //Resets angle if it
is greter than 2PI

      if (total_angle>(2.0*PI)){
                  printf("ERROR Angle was read as greater than 360 or less
than -360 degrees");
                  exit(1);
                                                              //If user entered
an angle too large, prints error
```

```
	}
	if (fabs(total_angle) >=0 && fabs(total_angle) <(PI/2)){
			circle->center.x=distance2*cos((total_angle));
			circle->center.y=distance2*sin((total_angle));
					//For new angle in quadrant 1
	}

	if (fabs(total_angle) >= (PI/2 && fabs(total_angle) <(PI))){
			angle=PI-total_angle;
			circle->center.x=-(distance2*cos(angle));

			circle->center.y=(distance2*sin(angle));
					//For new angle in quadrant
2
	}

	if (fabs(total_angle) >= PI && fabs(total_angle)<(3*PI/2)){
			angle=total_angle-PI;
			circle->center.x=-(distance2*cos(angle));

			circle->center.y=-(distance2*sin(angle));
					//For new angle in quadrant
3
	}

	if (fabs(total_angle) >= (3*PI/2) && fabs(total_angle) <(2*PI)){
			angle=(2*PI)-total_angle;
			circle->center.x=(distance2*cos(angle));

			circle->center.y=-(distance2*sin(angle));
					//For new angle in quadrant
4
	}

	circle->center.x=(circle->center.x)+delta.x;
	circle->center.y=(circle->center.y)+delta.y;
				//Returns center of entire shape to
orinal position

	return 0;
}

/*
**************************************************************************
**************************************************************************
******************
Polyline Functions created by Nathan Christian
**************************************************************************
**************************************************************************
******************
*/
int polyline_move(Polyline *polyline, double theta, struct Point delta)
{
	double total_angle, angle, distance2;
```

```
      distance2=sqrt(pow((polyline->vertex->x-
delta.x),2)+pow(polyline->vertex->y-delta.y,2));//Distance function was
not compatible with ptr Point.

      polyline->vertex->x=(polyline->vertex->x)-delta.x;
      polyline->vertex->y=(polyline->vertex->y)-delta.y;
                                    //Shifts center of shapes to origin for
easy computations

      if (polyline->vertex->x>0 && polyline->vertex->y>0)

      total_angle=theta+atan(polyline->vertex->y/polyline->vertex->x);
                        //Find new angle If initial angle begins in
quadrant 1

      if (polyline->vertex->x<0 && polyline->vertex->y>0)
                  total_angle=PI+theta-
(atan(fabs(polyline->vertex->y/polyline->vertex->x)));
      //Find new angle If initial angle begins in quadrant 2

      if (polyline->vertex->x<0 && polyline->vertex->y<0)

      total_angle=PI+theta+(atan(polyline->vertex->y/polyline->vertex->x))
;                     //Find new angle If initial angle begins in
quadrant 3

      if (polyline->vertex->x>0 && polyline->vertex->y<0)
                  total_angle=(2*PI)+theta-
(atan(fabs(polyline->vertex->y/polyline->vertex->x)));         //Find new
angle If initial angle begins in quadrant 4

      if (total_angle>(2.0*PI))
                              total_angle =total_angle-(2.0*PI);
                                                    //Resets angle if it
is greter than 2PI

      if (total_angle>(2.0*PI)){
                  printf("ERROR Angle was read as greater than 360or less
than -360 degrees");
                  exit(1);
                                                              //If user entered
an angle too large, prints error
      }
      if (fabs(total_angle) >=0 && fabs(total_angle) <(PI/2)){
                  polyline->vertex->x=distance2*cos(fabs(total_angle));
                  polyline->vertex->y=distance2*sin(fabs(total_angle));
                                          //If angle shifts into
quadrant 1
      }
      if (fabs(total_angle) >= (PI/2 && fabs(total_angle) <(PI))){
                  angle=PI-total_angle;
                  polyline->vertex->x=-(distance2*cos(angle));
```

```
                polyline->vertex->y=(distance2*sin(angle));
                                                //For new angle in quadrant
2
        }
        if (fabs(total_angle) >= PI && fabs(total_angle) <(3*PI/2)){
                angle=total_angle-PI;
                polyline->vertex->x=-(distance2*cos(angle));

                polyline->vertex->y=-(distance2*sin(angle));
                                        //For new angle in quadrant 3
        }
        if (fabs(total_angle) >= (3*PI/2) && fabs(total_angle) <(2*PI)){
                angle=(2*PI)-total_angle;
                polyline->vertex->x=(distance2*cos(angle));

                polyline->vertex->y=-(distance2*sin(angle));
                                        //For new angle in quadrant 4
        }
        polyline->vertex->x=(polyline->vertex->x)+delta.x;
        polyline->vertex->y=(polyline->vertex->y)+delta.y;
                                    //Returns shapes to original center.

        return 0;
}
void polyline_boundary(Polyline *polyline, Point *lower, Point *upper)
{
        int i;

        lower->x=polyline->vertex->x;
        lower->y=polyline->vertex->y;

        upper->x=polyline->vertex->x;
        upper->y=polyline->vertex->y;
                                                //Assigns lower/higher
values to polyline

        for(i=0;i<polyline->npoints;i++){
                if (lower->x>polyline->vertex->x)
                        lower->x=polyline->vertex->x;
                if (lower->y>polyline->vertex->y)
                        lower->y=polyline->vertex->y;
                if (upper->x<polyline->vertex->x)
                        upper->x=polyline->vertex->x;
                if (upper->y<polyline->vertex->y)
                        upper->y=polyline->vertex->y;
                                                        //scans line for
highest/lowest coordinates
        }
}
/*
**************************************************************************
**************************************************************************
******************
Polygon Functions created by Nathan Christian
```

```
**************************************************************************
**************************************************************************
******************
*/
int polygon_move(Polygon *polygon, double theta, struct Point delta)
{
      double total_angle, angle, distance2;

      distance2=sqrt(pow((polygon->vertex->x-
delta.x),2)+pow(polygon->vertex->y-delta.y,2));//Distance function not
compatible with ptr Point type
      polygon->vertex->x=(polygon->vertex->x)-delta.x;
      polygon->vertex->y=(polygon->vertex->y)-delta.y;
                              //Shifts center of shapes to origin for
easy computations

      if (polygon->vertex->x>0 && polygon->vertex->y>0)

      total_angle=theta+atan(polygon->vertex->y/polygon->vertex->x);
                        //Find new angle If initial angle begins in
quadrant 1

      if (polygon->vertex->x<0 && polygon->vertex->y>0)
                  total_angle=PI+theta-
(atan(fabs(polygon->vertex->y/polygon->vertex->x)));             //Find
new angle If initial angle begins in quadrant 2

      if (polygon->vertex->x<0 && polygon->vertex->y<0)

      total_angle=PI+theta+(atan(polygon->vertex->y/polygon->vertex->x));
                  //Find new angle If initial angle begins in quadrant 3

      if (polygon->vertex->x>0 && polygon->vertex->y<0)
                  total_angle=(2*PI)+theta-
(atan(fabs(polygon->vertex->y/polygon->vertex->x)));       //Find new
angle If initial angle begins in quadrant 4


      if (total_angle>(2.0*PI))
                              total_angle =total_angle-(2.0*PI);
                                                  //Resets angle if it is
greter than 2PI

      if (total_angle>(2.0*PI)){
                  printf("ERROR Angle was read as greater than 360or less
than -360 degrees");
                  exit(1);
                                                            //If user entered an
angle too large, prints error
      }

      if (fabs(total_angle) >=0 && fabs(total_angle) <(PI/2)){
                  polygon->vertex->x=distance2*cos(total_angle);
```

```
                polygon->vertex->y=distance2*sin(total_angle);
                                        //For new angle in quadrant 1

        }

        if (total_angle >= (PI/2 && total_angle <(PI))){
                angle=PI-total_angle;
                polygon->vertex->x=-(distance2*cos(angle));

                polygon->vertex->y=(distance2*sin(angle));
                                        //For new angle in quadrant 2

        }

        if (fabs(total_angle) >= PI && total_angle <(3*PI/2)){
                angle=total_angle-PI;
                polygon->vertex->x=-(distance2*cos(angle));

                polygon->vertex->y=-(distance2*sin(angle));
                                        //For new angle in quadrant 3

        }

        if (fabs(total_angle) >= (3*PI/2) && total_angle <(2*PI)){
                angle=(2*PI)-total_angle;
                polygon->vertex->x=(distance2*cos(angle));

                polygon->vertex->y=-(distance2*sin(angle));
                                        //For new angle in quadrant 1
        }

        polygon->vertex->x=(polygon->vertex->x)+delta.x;
        polygon->vertex->y=(polygon->vertex->y)+delta.y;
                                //Returns shape center to orinal position.

        return 0;
}
void polygon_boundary(Polygon *polygon, Point *lower, Point *upper)
{
        int i;
                lower->x=polygon->vertex->x;
                lower->y=polygon->vertex->y;
                upper->x=polygon->vertex->x;
                upper->y=polygon->vertex->y;
                                        //Assigns lower/higher values to
polyline

                for(i=0;i<polygon->npoints;i++){
                        if (lower->x>polygon->vertex->x)
                              lower->x=polygon->vertex->x;
                        if (lower->y>polygon->vertex->y)
                              lower->y=polygon->vertex->y;
                        if (upper->x<polygon->vertex->x)
                              upper->x=polygon->vertex->x;
```

```
                              if (upper->y<polygon->vertex->y)
                                    upper->y=polygon->vertex->y;
                                                //scans line for
highest/lowest coordinates
                  polygon->vertex++;
                  }
}

/*
**************************************************************************
**************************************************************************
******************
Arc Functions created by Nathan Christian
**************************************************************************
**************************************************************************
******************
*/
int arc_move(Arc *arc, double theta, double distance2, struct Point delta)
{
      int n;
      double total_angle, angle;

      arc->center.x=(arc->center.x)-delta.x;
      arc->center.y=(arc->center.y)-delta.y;
                                    //Shifts center of shapes to origin
for easy computations

      if (arc->center.x>0 && arc->center.y>0)
                  total_angle=theta+atan(arc->center.y/arc->center.x);
                                    //Find new angle if initial angle
begins in quadrant 1

      if (arc ->center.x<0 && arc->center.y>0)
                  total_angle=PI+theta-
(atan(fabs(arc->center.y/arc->center.x)));
//Find new angle if inigial angle begins in quadrant 2

      if (arc ->center.x<0 && arc->center.y<0)
                  total_angle=PI+theta+(atan(arc->center.y/arc->center.x));
                                 //Find new angle if initial angle begins in
quadrant 3

      if (arc->center.x>0 && arc->center.y<0)
                  total_angle=(2*PI)+theta-
(atan(fabs(arc->center.y/arc->center.x)));                        //Find
new angle if initial angle begins in quadrant 4

      if (fabs(total_angle)>(2.0*PI))
                              total_angle =total_angle-(2.0*PI);
                                                //Resets angle if it is
greter than 2PI

      if (fabs(total_angle)>(2.0*PI)){
```

```
                printf("ERROR Angle was read as greater than 360or less
than -360 degrees");
                exit(1);
                                                        //Prints error if
user selected an angle too large.
      }

      if (fabs(total_angle) >=0 && total_angle <(PI/2)){
                arc->center.x=distance2*cos(fabs(total_angle));
                arc->center.y=distance2*sin(fabs(total_angle));
                                  //For mew angle in quadrant 1
      }

      if (fabs(total_angle) >= (PI/2) && fabs(total_angle) <(PI)); {
                angle=PI-total_angle;
                arc->center.x=-(distance2*cos(angle));

                arc->center.y=(distance2*sin(angle));
                                        //For mew angle in quadrant 2
      }

      if (fabs(total_angle) >= PI && fabs(total_angle) <(3*PI/2)){
                angle=total_angle-PI;
                arc->center.x=-(distance2*cos(angle));

                arc->center.y=-(distance2*sin(angle));
                                        //For mew angle in quadrant 3
      }

      if (fabs(total_angle) >= (3*PI/2) && fabs(total_angle) <(2*PI)){
                angle=(2*PI)-total_angle;
                arc->center.x=(distance2*cos(angle));

                arc->center.y=-(distance2*sin(angle));
                                        //For mew angle in quadrant 4
      }

      for (n=0;n<3;n++)
                arc->angle[n]=arc->angle[n]+(theta*180/PI);
                                        //Rotate arc to match new angle

      arc->center.x=(arc->center.x)+delta.x;
      arc->center.y=(arc->center.y)+delta.y;
                                    //Returns shape to original center.

      return 1;
}
void arc_boundary(Arc *arc, Point *lower, Point *upper)
{
      int i;
      Point a1;
      lower->y=lower->x=1000000;
```

```
      upper->x=upper->y=-1000000;
                                          //Initializes upper/lower
boundaries


                  for (i=0;i<3;i++){
                              arc->angle[i]=arc->angle[i]*M_PI/180;
                                                //Convert arc angle to
radians for rotation


      a1.x=arc->center.x+((arc->radius)*cos(arc->angle[i]));

      a1.y=arc->center.y+((arc->radius)*sin(arc->angle[i]));
                    //Finds upper and lower boundary of box around arc

                              if (upper->x<a1.x)
                                    upper->x=a1.x;
                              if (lower->x>a1.x)
                                    lower->x=a1.x;
                              if (upper->y<a1.y)
                                    upper->y=a1.y;
                              if (lower->y>a1.y)
                                    lower->y=a1.y;
                                                      //Finds
highest/lowest box in this line
                  }

                  if (lower->x>arc->center.x)
                              lower->x=arc->center.x;
                  if (lower->y>arc->center.y)
                              lower->y=arc->center.y;
                  if (upper->x<arc->center.x)
                              upper->x=arc->center.x;
                  if (upper->y<arc->center.y)
                              upper->y=arc->center.y;
                                                      //Compares
highest/lowest part of box to center point
}
/*
**************************************************************************
**************************************************************************
******************
Functions Provided by Dr. Kim
**************************************************************************
**************************************************************************
******************
*/
double polyline_length(Polyline *polyline)
{
    assert(polyline != NULL);
    assert(polyline->npoints >= 3);
    int i;
    double len;
```

```
    len = 0.0;
    for (i=1; i<polyline->npoints; i++)
        len += distance(polyline->vertex[i-1], polyline->vertex[i]);

    return len;
}
double polygon_length(Polygon *polygon)
{
    assert(polygon != NULL);
    assert(polygon->npoints >= 3);
    int i;
    double len;

    len = distance(polygon->vertex[0], polygon->vertex[polygon->npoints-
1]);
    for (i=1; i<polygon->npoints; i++)
        len += distance(polygon->vertex[i-1], polygon->vertex[i]);

    return len;
}
```