```
/*******************************************************************************************
 GeoDraw Version 1.0 (23.10) - Last revision: 01/10/2023
 - A simple C/C++ 2D drawing library that outputs drawings as JavaScript
 - Includes a simplified C function interface

 NOTICE: Copyright (C) 2023 Conor McArdle, Dublin City University - All Rights Reserved.
 Permission is granted to students registered for DCU module EE219: C/C++ for Engineers
 to use this code for their personal use only. No permission is granted to share this file
 by any means, including but not limited to, uploading it or submitting it to any private
 or public server on the Web/Internet or any online service which publishes or otherwise
 processes or stores uploaded content. This notice must stay attached to this file.
*******************************************************************************************/

#ifndef GEODRAW_H_INCLUDED
#define GEODRAW_H_INCLUDED

#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <string>
#include <vector>
#include <math.h>
#include <bits/stdc++.h>

namespace GeoDraw {

using std::cout;
using std::cerr;
using std::endl;
using std::string;
using std::vector;
using std::ofstream;

extern string _gd_html_pre;  // html to appear before canvas in output web page
extern string _gd_html_post; // html to appear after canvas in output web page


/*******************************************************************************************
 Utility methods and defines
*******************************************************************************************/
typedef unsigned int u_int;
const double PI = acos(-1);
string gd_to_string(int a);
string gd_to_string(double a, unsigned short decimalPlaces);


/*******************************************************************************************
```

```
 FillState, Font, TextAlign and Color class
*********************************************************************************************/
enum FillState { FILLED, UNFILLED };

enum Font { Arial, Courier, Times };

enum TextAlign { LEFT, CENTER, RIGHT };

class Color {
private:
   double _r, _g, _b;
   string _24BitColor;
public:
   Color(double r, double g, double b) : _r(r), _g(g), _b(b) {
      _24BitColor = "(";
      _24BitColor += gd_to_string((int)round(_r * 255)) + ",";
      _24BitColor += gd_to_string((int)round(_g * 255)) + ",";
      _24BitColor += gd_to_string((int)round(_b * 255)) + ")";
   }
   double r() const { return _r; }
   double g() const { return _g; }
   double b() const { return _b; }
   string to24BitColorString() const { return _24BitColor; }
};


/*********************************************************************************************
 Pre-defined Colors
*********************************************************************************************/
const Color BLACK(0,0,0);
const Color WHITE(1,1,1);
const Color LIGHT_GRAY(0.8,0.8,0.8);
const Color GRAY(0.5,0.5,0.5);
const Color DARK_GRAY(0.2,0.2,0.2);
const Color LIGHT_RED(1,0.5,0.5);
const Color RED(1,0,0);
const Color DARK_RED(0.5,0,0);
const Color LIGHT_GREEN(0.5,1,0.5);
const Color GREEN(0,1,0);
const Color DARK_GREEN(0,0.5,0);
const Color LIGHT_BLUE(0.5,0.5,1);
const Color BLUE(0,0,1);
const Color DARK_BLUE(0,0,0.5);
const Color YELLOW(1,1,0);
const Color TEAL(0,0.5,0.5);
const Color CYAN(0,1,1);
const Color MAGENTA(1,0,1);
const Color PINK(1,0.5,1);
const Color ORANGE(1,0.5,0);


/*********************************************************************************************
 Forward Declaratoins
*********************************************************************************************/
class LineSeg;
class Circle;
class Polygon;
class Geometry;
class GeometryList;
class Canvas;


/*********************************************************************************************
 Coord class
*********************************************************************************************/
class Coord {
private:
   double _x, _y;
public:
   Coord(double x, double y) : _x(x), _y(y) {}
   double x() const { return _x; }
   double y() const { return _y; }
   void moveBy(double x, double y) { _x += x; _y += y; }
   void rotate(double angle) {
      double tmp_x = _x * cos(angle) - _y * sin(angle);
      double tmp_y = _x * sin(angle) + _y * cos(angle);
      _x = tmp_x;
      _y = tmp_y;
   }
   void rotate(double angle, double x, double y) {
      moveBy(-x, -y);
      rotate(angle);
      moveBy(x, y);
   }
   void scale(double factor) { _x *= factor; _y *= factor; }
   double distanceTo(const Coord & otherCoord) const {
      double x_diff = _x - otherCoord._x;
      double y_diff = _y - otherCoord._y;
      return sqrt(x_diff*x_diff + y_diff*y_diff);
   }
   string toString() const {
      return gd_to_string(_x) + "," + gd_to_string(_y);
   }
};
```

```
/**********************************************************************************************
 CoordList class
**********************************************************************************************/
class CoordList {
private:
   vector<Coord> coords;
public:
   void add(double x, double y) {
      coords.push_back(Coord(x,y));
   }
   void add(const Coord & c) {
      coords.push_back(c);
   }
   Coord & operator [] (u_int i) {
      if ((coords.size() == 0) || (i > coords.size()-1))
         cerr << "GeoDraw::CoordList::[] - error : index out of bounds at index " << i << endl;
      return coords[i];
   }
   u_int size() {
      return coords.size();
   }
   void clear() {
      coords.clear();
   }
};


/**********************************************************************************************
 CanvasElement class
**********************************************************************************************/
class CanvasElement {
   friend class Canvas;
private:
   u_int pauseAfter;  // milliseconds to pause after drawing this canvas element
   virtual CanvasElement * clone() const = 0;
   virtual string toJavaScript() const = 0;
protected:
   Color color = BLACK;
   u_int penWidth = 1;
   FillState fillState = UNFILLED;
   CanvasElement() : pauseAfter(0), color(BLACK), penWidth(1), fillState(UNFILLED) {}
public:
   virtual ~CanvasElement() {}
};


/**********************************************************************************************
 Abstract base class for all geometry objects
```

```
*******************************************************************************************/
class Geometry : public CanvasElement {
   friend class GeometryList;
public:
   Geometry() {}
   virtual ~Geometry() {};
   /// Geometry object public interface methods
   virtual Geometry & moveBy(double x, double y) = 0;
   virtual Geometry & rotate(double angle, double x, double y) = 0;
   virtual Geometry & rotate(double angle, const Coord & cRef) = 0;
   virtual Geometry & scale(double factor) = 0;
private:
   virtual Geometry * clone() const = 0;
};


/*******************************************************************************************
 A list of Geomtry objects
*******************************************************************************************/
class GeometryList {
private:
   vector<Geometry *> geoms;
public:
   void add(const Geometry & geom) {
      geoms.push_back(geom.clone());
   }
   Geometry & operator [] (u_int i) {
      if ((geoms.size() == 0) || (i > geoms.size()-1))
         cerr << "GeoDraw::GeometryList::[] - error : index out of bounds at index " << i <<
endl;
      return *geoms[i];
   }
   u_int size() {
      return geoms.size();
   }
   void clear() {
      geoms.clear();
   }
   virtual ~GeometryList() {
      for (u_int i=0; i<geoms.size(); i++)
         if (geoms[i] == NULL)
            cerr << "GeoDraw::~GeometryList() - error : null pointer in destructor " << i <<
endl;
         else
            delete geoms[i];
   }
};
```

```
/********************************************************************************************
 Point class
*********************************************************************************************/
class Point : public Geometry {
private:
   Coord coord;
   string toJavaScript() const;
   string getType() const { return "Point"; }
   virtual Geometry * clone() const { return new Point(*this); }
public:
   Point(double x, double y) : coord(x,y) {}
   Point(const Coord & c) : coord(c) {}
   double x() { return coord.x(); }
   double y() { return coord.y(); }
   double distanceTo(const Point & p) {
      return coord.distanceTo(p.coord);
   }
   Coord coordinate() const {
      return Coord(coord.x(), coord.y());
   }
   /// Geometry base class method implementations/overrides
   Point & moveBy(double x, double y) { coord.moveBy(x,y); return *this; }
   Point & rotate(double angle, double x, double y) { coord.rotate(angle, x, y); return *this; }
   Point & rotate(double angle, const Coord & cRef) { coord.rotate(angle, cRef.x(), cRef.y());
return *this; }
   Point & scale(double factor) { coord.scale(factor); return *this; }
};


/********************************************************************************************
 LineSeg class
*********************************************************************************************/
class LineSeg : public Geometry {
private:
   Coord c1, c2;
   string toJavaScript() const;
   string getType() const { return "LineSeg"; }
   virtual Geometry * clone() const { return new LineSeg(*this); }
public:
   LineSeg(const Coord & _c1, const Coord & _c2) : c1(_c1), c2(_c2) {}
   LineSeg(double x1, double y1, double x2, double y2) : c1(x1,y1), c2(x2,y2) {}
   CoordList asCoordList() const {
      CoordList coords;
      coords.add(c1);
      coords.add(c2);
      return coords;
   }
```

```
    /// Geometry base class method implementations/overrides
    LineSeg & moveBy(double x, double y) {
        c1.moveBy(x,y);
        c2.moveBy(x,y);
        return *this;
    }
    LineSeg & rotate(double angle, double x, double y) {
        c1.rotate(angle,x,y);
        c2.rotate(angle,x,y);
        return *this;
    }
    LineSeg & rotate(double angle, const Coord & cRef) {
        return this->rotate(angle, cRef.x(), cRef.y());
    }
    LineSeg & scale(double factor) {
        c1.scale(factor);
        c2.scale(factor);
        return *this;
    }
};


/*********************************************************************************************
 Circle class
*********************************************************************************************/
class Circle : public Geometry {
private:
    Coord cen; double radius;
    string toJavaScript() const;
    string getType() const { return "Circle"; }
    virtual Geometry * clone() const { return new Circle(*this); }
public:
    Circle(const Coord & c, double r) : cen(c), radius(r) {}
    Circle(double c_x, double c_y, double r) : cen(c_x,c_y), radius(r) {}
    /// Geometry base class method implementations/overrides
    Circle & moveBy(double x, double y) {
        cen.moveBy(x,y);
        return *this;
    }
    Circle & rotate(double angle, double x, double y) {
        cen.rotate(angle,x,y);
        return *this;
    }
    Circle & rotate(double angle, const Coord & cRef) {
        return this->rotate(angle, cRef.x(), cRef.y());
    }
    Circle & scale(double factor) {
        radius *= factor;
        return *this;
    }
};


/*******************************************************************************************
 Polygon class
*******************************************************************************************/
class Polygon : public Geometry {
private:
    vector<Coord> vertices;
    string toJavaScript() const;
    string getType() const { return "Polygon"; }
    virtual Geometry * clone() const { return new Polygon(*this); }
public:
    Polygon() {}
    Polygon(const vector<Coord> & coords) : vertices(coords) { }
    Polygon(CoordList & coords) {
        for (u_int i=0; i<coords.size(); i++)
            vertices.push_back(coords[i]);
    }
    Polygon(const Coord coords[], u_int size) {
        for (u_int i=0; i<size; i++)
            vertices.push_back(coords[i]);
    }
    void add(Coord c) { vertices.push_back(c); }
    void add(double x, double y) { vertices.push_back(Coord(x,y)); }
    u_int size() { return vertices.size(); }
    CoordList asCoordList() const {
        CoordList coords;
        for (u_int i=0; i<vertices.size(); i++)
            coords.add(vertices[i]);
        return coords;
    }
    /// Geometry base class method implementations/overrides
    Polygon & moveBy(double x, double y) {
        for (u_int i=0; i<vertices.size(); i++)
            vertices[i].moveBy(x,y);
        return *this;
    }
    Polygon & rotate(double angle, double x, double y) {
        for (u_int i=0; i<vertices.size(); i++)
            vertices[i].rotate(angle,x,y);
        return *this;
    }
    Polygon & rotate(double angle, const Coord & cRef) {
        for (u_int i=0; i<vertices.size(); i++)
            vertices[i].rotate(angle, cRef.x(), cRef.y());
```

```
      return *this;
   }
   Polygon & scale(double factor) {
      for (u_int i=0; i<vertices.size(); i++)
         vertices[i].scale(factor);
      return *this;
   }
};


/*********************************************************************************************
 Text class
*********************************************************************************************/
class Text : public CanvasElement {
private:
   string text;
   Coord position;
   Font font;
   u_int fontSize;
   TextAlign alignment;
   string toJavaScript() const;
   virtual Text * clone() const { return new Text(*this); }
public:
   Text(const string & txt, Coord pos) : text(txt), position(pos), font(Arial), fontSize(20),
alignment(LEFT) { fillState = FILLED; }
   Text(const string & txt, double x, double y) : text(txt), position(x,y), font(Arial), fontSize(20),
alignment(LEFT) { fillState = FILLED; }
   Text(const string & txt, Coord pos, Font fnt, u_int font_size, TextAlign algn = LEFT) :
text(txt), position(pos), font(fnt), fontSize(font_size), alignment(algn) { fillState = FILLED; }
   Text(const string & txt, double x, double y, Font fnt, u_int font_size, TextAlign algn = LEFT)
: text(txt), position(x,y), font(fnt), fontSize(font_size), alignment(algn)  { fillState = FILLED; }
};


/*********************************************************************************************
 Canvas class
*********************************************************************************************/
class Canvas {

private:

   vector<CanvasElement *> elements;    // container for canvas drawing elements
   u_int _xDim;                         // canvas size in X direction
   u_int _yDim;                         // canvas size in Y direction
   Color bg_color;                      // canvas background color
   string outFileName;                  // output file name
   Color default_pen_color;             // default color of outline for non-filled shapes
   Color default_fill_color;            // default color of filled shapes
   u_int default_pen_width;          // default width of pen, in pixels

public:
   Canvas(u_int x=600, u_int y=600, string outFile = "MyDrawing.html", Color color =
WHITE) :
      _xDim(x),
      _yDim(y),
      bg_color(color),
      outFileName(outFile),
      default_pen_color(BLACK),
      default_fill_color(GRAY),
      default_pen_width(2) { }

   unsigned int xDim() { return _xDim; }

   unsigned int yDim() { return _yDim; }

   void setOutFileName(string fileName) { outFileName = fileName; }

   Color penColor() { return default_pen_color; }

   Color fillColor() { return default_fill_color; }

   u_int penWidth() { return default_pen_width; }

   void setBackgroundColor(Color clr) { bg_color = clr; }

   void setPenColor(Color clr) { default_pen_color = clr; }

   void setPenWidth(u_int width) { default_pen_width = width > 0 ? default_pen_width : 1; }

   void setFillColor(Color clr) { default_fill_color = clr; }

   void add(const CanvasElement & elem) { elements.push_back(elem.clone()); }

   void add(const CanvasElement & elem, const Color & color, FillState is_filled =
UNFILLED) {
      CanvasElement* elemPtr = elem.clone();
      elemPtr->color = color;
      elemPtr->fillState = is_filled;
      elements.push_back(elemPtr);
   }

   void add(const CanvasElement & elem, const Color & color, u_int pen_width) {
      CanvasElement* elemPtr = elem.clone();
      elemPtr->color = color;
      elemPtr->penWidth = pen_width;
      elements.push_back(elemPtr);
   }

   void add(const CanvasElement & elem, u_int pen_width, const Color & color) {
      CanvasElement* elemPtr = elem.clone();
      elemPtr->color = default_pen_color;
      elemPtr->penWidth = pen_width;
      elements.push_back(elemPtr);
   }

   void add(GeometryList & geomList) {
      for (u_int i=0; i<geomList.size(); i++)
         add(geomList[i]);
   }

   void add(GeometryList & geomList, const Color & color, FillState is_filled = UNFILLED) {
      for (u_int i=0; i<geomList.size(); i++)
         add(geomList[i], color, is_filled);
   }

   void add(GeometryList & geomList, const Color & color, u_int pen_width) {
      for (u_int i=0; i<geomList.size(); i++)
         add(geomList[i], color, pen_width);
   }

   void pause(u_int delay_ms) { // pause after after last canvas element added
      if (elements.size() == 0) return;
      elements[elements.size()-1]->pauseAfter += delay_ms;
   }

   void draw(); // send all drawing elements to HTML/JS file

   void draw(string filename); // draw to specified file

   void clear() {
      Polygon p;
      p.add(Coord(0, 0));
      p.add(Coord(_xDim, 0));
      p.add(Coord(_xDim, _yDim));
      p.add(Coord(0, _yDim));
      this->add(p, bg_color , FILLED);
   }

   ~Canvas() {
      for (u_int i=0; i<elements.size(); i++)
         if (elements[i] != NULL)
            delete elements[i];
   }
```

```
private:

    string generateJSDrawingString();  // helper method for Canvas::draw()
};


} // end GeoDraw namespace


/*********************************************************************************************
*
*   SIMPLIFIED C-STYLE INTERFACE for GeoDraw - Function Declarations
*
*********************************************************************************************/

using namespace GeoDraw;

// Returns the width (x direction) of the drawing canvas (number of pixels)
u_int  gd_getCanvasSizeX ();

// Returns the height (y direction) of the drawing canvas (number of pixels)
u_int  gd_getCanvasSizeY ();

// Change the canvas size. Side-effect: clears and resets the canvas.
void   gd_resetCanvasSize (u_int xSize, u_int ySize);

// Set the canvas background colour with a standard colour
// NOTE: Canvas colour will not change until gd_clear() is called
// The available colours are:
// BLACK, WHITE, LIGHT_GRAY, GRAY, DARK_GRAY,
// LIGHT_RED, RED, DARK_RED, LIGHT_GREEN, GREEN, DARK_GREEN,
// LIGHT_BLUE, BLUE, DARK_BLUE, YELLOW, TEAL, CYAN, MAGENTA,
// PINK, ORANGE.
// Example Usage: gd_setCanvasColor(DARK_GRAY);
void   gd_setCanvasColor (Color color);

// Set the canvas background colour with a custom (red,green,blue) colour
// Colour components have values in the range 0.0 to 1.0
// For example (1.0,0.0,0.0) is red
void   gd_setCanvasColor (double r, double g, double b);

// Set the width of the drawing pen (number of pixels)
void   gd_setPenWidth (u_int width);

// Set the drawing pen colour with a standard colour (see colour list above)
// Example Usage: gd_setPenColor(BLUE);
void   gd_setPenColor (Color color);
```

```
// Set the drawing pen colour with a custom (red,green,blue) colour
// Colour components have values in the range 0.0 to 1.0
// For example, (0.0,1.0,0.0) is green
void   gd_setPenColor (double r, double g, double b);

// Set the colour for filled shapes with a standard colour (see colour list above)
// Example Usage: gd_setFillColor(GREEN);
void   gd_setFillColor (Color color);

// Set the colour for filled shapes with a custom (red,green,blue) colour
// Colour components have values in the range 0.0 to 1.0
// For example, (0.0,0.0,1.0) is blue
void   gd_setFillColor (double r, double g, double b);

// Set the font to use when drawing text
// Options are: Arial, Courier, Times
// Example Usage: gd_setFont(Courier);
void   gd_setFont (Font font);

// Set the height of text characters, in units of pixels
void   gd_setTextSize (u_int font_size);

// Set the text colour with a standard colour (see colour list above)
// Example Usage: gd_setTextColor(ORANGE);
void   gd_setTextColor (Color color);

// Set the text colour with a custom (red,green,blue) colour
// Colour components have values in the range 0.0 to 1.0
// For example, (0.0,0.0,1.0) is blue
void   gd_setTextColor (double r, double g, double b);

// Set the text alignment relative to the coordinate at which text is drawn
// Options are: LEFT, CENTER, RIGHT
// Example Usage: gd_setTextAlignment(CENTER);
void   gd_setTextAlignment (TextAlign alignment);

// Draw a point at the specified coordinate
// The last set pen width and colour will be used
void   gd_point (double x, double y);

// Draw a line segmented between points (x1,y1) and (x2,y2)
// The last set pen width and colour will be used
void   gd_line (double x1, double y1, double x2, double y2);

// Draw a circle centred at (x,y) with the specified radius
// The last set pen width and colour will be used
void   gd_circle (double x, double y, double radius);
```

```
// Draw a filled circle centred at (x,y) with the specified radius
// The last set fill colour will be used
void   gd_circleFilled (double x, double y, double radius);

// Draw a triangle with the specified vertices (x1,y1),(x2,y2),(x3,y3)
// The last set pen width and colour will be used
void   gd_triangle (double x1, double y1, double x2, double y2,
                   double x3, double y3);

// Draw a filled triangle with the specified vertices (x1,y1),(x2,y2),(x3,y3)
// The last set fill colour will be used
void   gd_triangleFilled (double x1, double y1, double x2, double y2,
                         double x3, double y3);

// Draw a quadrilateral with the specified vertices (x1,y1),(x2,y2),(x3,y3),(x4,y4)
// The last set pen width and colour will be used
void   gd_quad (double x1, double y1, double x2, double y2,
               double x3, double y3, double x4, double y4);

// Draw a filled quadrilateral with the specified vertices (x1,y1),(x2,y2),(x3,y3),(x4,y4)
// The last set fill colour will be used
void   gd_quadFilled (double x1, double y1, double x2, double y2,
                     double x3, double y3, double x4, double y4);

// Draw text at the specified (x,y) coordinate
// The last set font, text size, text colour and text alignment will be used
// txt is a C++ string that may be passed a string literal, such as "Hello World!"
void   gd_text (string txt, double x, double y);

// Pause for a given number of milliseconds before displaying the next drawing element
added
// after calling gd_pause(). Example Usage:
// gd_circle(100,100,50);
// gd_pause(1000); // pause 1 second before next circle is drawn
// gd_circle(200,200,20);
void   gd_pause (u_int pauseTimeMs);

// Clear the drawing canvas. The canvas will be repainted
// with the last set background colour (see gd_setCanvasColor())
void   gd_clear ();

// Write the canvas drawing with all previous draw and pause commands to file.
// The output file format is HTML with embedded JavaScript.
// The filename should be given a .html extension so it can be opened easily in a web
browser.
// Note: saving does not clear the canvas state.
void   gd_save (string filename);
```

```
#endif // GEODRAW_H_INCLUDED
```