C++ Exercise 6: Shapes and Transformations

Deadline: Monday, October 26, 2020 by 23:59:59 EET Timezone.
How to submit: Check your code into your GitHub repository, the default `master` branch, tag it as `ex06submit` (all lowercase, no dashes).
Grading: This exercise is worth 3‰(or 3%) of the total grade.
Objectives: Develop a software that receives a list of commands on separate lines that define shapes (either regular n-gons or circles), apply transformations to those shapes, copy and group the shapes and finally paint them as SVG.

Two Stages

It is suggested that you do this lab in two stages. Each stage will have unit tests, so that you can check yourself. You are welcome to use other implementations as well; so you can also ignore this suggestion, if you know a better way.

1. During the first stage you check if you can create some basic shape (for example a polygon), apply one or more transformations to that basic shape and draw the shape as an SVG. During this stage you can try out, if you can create transformation matrices of all kinds and understand the SVG output.

2. During the second stage you can create groups of shapes and see, if you can transform and draw entire groups (and also nested groups of groups) using the *Composite* design pattern and virtual functions `transform()` and `draw()`. You will have abstract `Shape` class, and also its child class `Group` that can contain other shapes (circles, polygons and other groups).

Description

Assume that we need to draw groups of simple shapes such as circles, rectangles, triangles (and some other polygons such as stars). We start from just two kinds of "basic shapes" (circles and regular polygons) – they can be created in any color by specifying their RGB color coordinates. We can transform these shapes by multiplying their vertices $(x, y, 1)$ by 3×3 transformation matrices (to support translations, rotations, scaling and sharing). We can also group shapes together: Each group can consist of basic shapes (after they have received one or or more transformations), and also a group can obtain other groups.
This editing process is similar to what you would do in a graphics editor (you can resize, drag or rotate shapes and combine them in groups on multiple levels). After you have grouped together the shapes you want, they are
Then they are output to a vector graphics format (SVG). We want to do all coordinate transformations (such as translating the coordinates) inside our C++ program using matrix multiplication. (I.e. we avoid any coordinate transformation operations in the SVG itself; it only displays closed paths from points that are already transformed.)

Transformations

There are only two types of basic shapes that can be created: it is either a unit circle (circle with center in $(0, 0)$ and radius 1) or a regular n-gon (where $n \geq 3$; one of the n-gon's vertices is always $(1, 0)$ - the point on the X axis and its center is point $(0, 0)$). See Figure 1

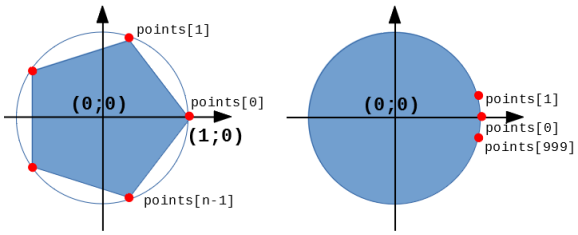


Figure 1: Polygon (here $n = 5$) and Circle before any transformations.

All the other shapes are obtained from these two basic shapes by applying some linear coordinate transformations. See `https://bit.ly/30Yq3xN` - Affine transformations for 2D images. There are just 4 transformations:

- Translation denoted by `TRA dx dx`: it shifts all points in a shape by a vector (dx, dy)
- Rotation denoted by `ROT alpha`: it rotates all points in a shape by angle `alpha` degrees around the center $(0, 0)$.
- Scaling denoted by `SCA cx cy`: it multiplies coordinates of all points in a shape by `cx` and `cy` respectively. The numbers `cx,cy` can be negative; we can also get mirror images of the original figure.
- Shearing denoted by `SHE cx cy`: it maps every point (x, y) to a point $(x + cx \cdot y, y + cy \cdot x)$.

Input and Output

Input files contain the following commands.

- Commands `CIRCLE #ABCDEF` or `POLYGON #ABCDEF n` push a new unit circle or a unit n-gon to the top of the stack; its color is defined by the RGB color coordinates (`#000000` is black, `#FF0000` is red and so on.).
- Commands `TRANSF TRA dx dy`, `TRANSF SCA cx cy`, etc. apply one of the linear transformations to the shape that is at the top of the stack.
- Command `COPY` takes the element at the top of the stack, makes a deep copy of it and pushes it to the top of the stack.
- Command `GROUP N` pops the top N elements off the stack, makes a group of these elements and pushes this as a group back to the stack.
- Command `SHOW width height` pops the top element off the stack and outputs an SVG image with size `width` by `height` pixels.

Limitations

You can assume the following:

1. The total number of shapes (basic shapes or groups) ever pushed to the stack does not exceed 1000.

2. No polygon will have more than 1000 sides.
3. Under transformations defined above, circles can be implemened as regular polygons with many vertices (for example, 1000 vertices). The test data will not apply extreme streching to those circles that would expose this workaround.
4. The width and the height is between 100 to 1000 pixels; borders of width 1 are visible.
5. Grouping in the input files does not cause exceptions (`pop()` on an empty stack).

Implementation Details

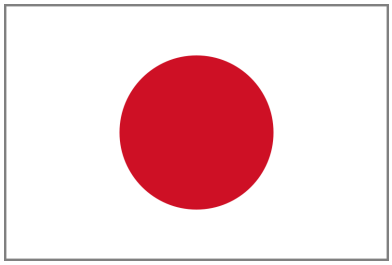
1. Implement whatever CPP and H files you need (most likely you will need `ShapesMain.cpp` as your main method, `Circle.cpp`, `Circle.h`, `Polygon.cpp`, `Polygon.h`, `Group.cpp`, `Group.h` (group of shapes), `Shape.cpp` (abstract shape parent for circles, polygons and groups), `Matrix.h` (to implement matrix multiplication for doubles; you can also store point coordinates as matrices, if you want), `Stack.cpp` (some stack implementation; you can use a large array; you can assume that the stack size never exceeds 1000 entries).
2. Inside the group you can store the children as an array of `Shape*` (pointers to shapes, including other groups).
3. Implement the composite pattern to apply geometric transformations for shapes, including nested shape groups.

Sample Input and Output

Sample input **test01in.txt**:

```
POLYGON #FFFFF 4
TRANSF ROT 45
TRANSF SCA 1.4142135 1.4143135
TRANSF SCA 150 100
TRANSF TRA 150 100
CIRCLE #BC002D
TRANSF SCA 60 60
TRANSF TRA 150 100
GROUP 2
SHOW 300 200
```

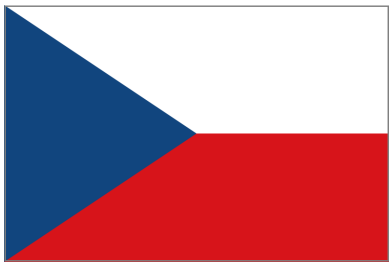
Expected output **test01expected.svg**:



Sample input **test02in.txt**:

```
POLYGON #FFFFFF 4
TRANSF ROT 45
TRANSF SCA 1.4142135 1.4143135
TRANSF TRA 0 -1
POLYGON #D7141A 4
TRANSF ROT 45
TRANSF SCA 1.4142135 1.4143135
TRANSF TRA 0 1
GROUP 2
TRANSF SCA 150 50
TRANSF TRA 150 100
POLYGON #11457E 3
TRANSF TRA 0.5 0
TRANSF SCA 115.4701 115.4701
TRANSF TRA 0 100
GROUP 2
SHOW 300 200
```

Expected output **test02expected.svg**:



Standard Output as an SVG File

You need to know just two SVG elements in order to complete this task. SVG is a rich vector-graphics language that has its own mechanisms to do coordinate transformations, but we are not using them here, since all the point coordinates will be computed in your C++ code.

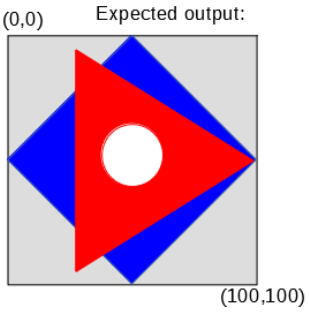
What is the painting order in SVG:

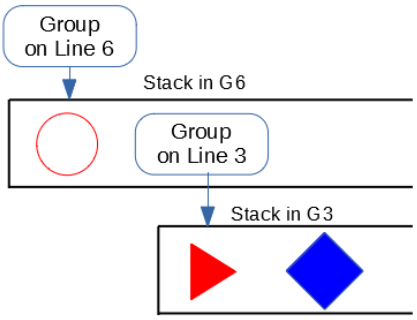
This exercise does not use special SVG parameters to affect the painting order (such as the CSS property `z-index`). By default the shapes (`path` elements) in SVG are painted on top of each other in the order in which they appear in the document you output. For the Japanese flag you would first paint the white rectangle and then the red circle.

If you use stacks in **two** places (Firstly, to accumulate the shapes received in the input, Secondly, to store child elements in your groups), the shape order is reversed twice. So they will be painted in the same order as you specify them in the input file.

In this example, the input file contains shapes in the order we want them stacked (first the blue square, then the red triangle, finally the white circle).

```
1 POLYGON #0000FF 4
2 POLYGON #FF0000 3
3 GROUP 2
4 CIRCLE #FFFFFF
5 TRANSF SCA 0.25 0.25
6 GROUP 2
7 TRANSF SCA 50 50
8 TRANSF TRA 50 50
9 SHOW 100 100
```





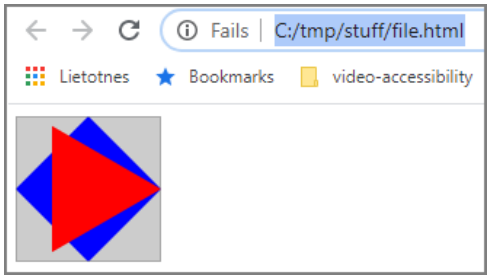
In the Group3 (created on input Line 3) red triangle is pushed into the Group3 first (as in the Line 3 it popped out the input stack), followed by the blue square. In the Group 6 (created on input Line 6), the white circle is pushed into its stack first; followed by Group3 that was

created before. As you call `draw()` function, they are drawn in the correct order again: first the blue square, then the red triangle, finally the white circle.

Testing your Code

Unlike many previous exercises it is not easy to check real-valued matrix multiplication results using Linux `diff` utility; the rounding errors are not always predictable. It is expected that different students can get slightly different outputs. For this reason the testing of this program is best done by inspecting the generated images.
One or more SVG outputs can be embedded in an HTML file:

```
1 <html>
2 <head>
3 <title>Embedded SVG</title>
4 </head>
5 <body>
6 <img src="image.svg"/>
7 </body>
8 </html>
```



The file `image.svg` is shown below. The "interesting" part of the SVG is on Lines 4 and 5 only - it draws two polygons. (The first two lines define a light-gray canvas of size 100×100. The last `<rect>` element draws a darker-gray rectangle around the image.)
All the polygons and circles need just one SVG element `path`. It has attribute `d` that has a long list of commands followed by coordinates (and also ability to specify the fill shape). For example, `M 100 50` moves the pencil to the specified coordinates $(100, 50)$; command `L 50 0` draws a line to a new point $(50, 0)$ and so on. Please note that the y axis in SVG goes downwards.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
3 <path d="M 0 0 L 100 0 L 100 100 L 0 100" fill="#cccccc"/>
4 <path d="M 100 50 L 50 0 L 0 50 L 50 100" fill="#0000ff"/>
5 <path d="M 100 50 L 25 93.3 L 25 6.7" fill="#ff0000"/>
6 <rect x="0" y="0" width="100" height="100"
7 stroke="#999999" fill="none" stroke-width="1"/>
8 </svg>
```

Note. You can draw the circles using same `path` command as well (treating them as polygons with 1000 vertices). Therefore you would need just the `path` element in SVG (and its commands "M" - move to, "L" - line to). (There are many other SVG elements to create advanced visualizations, but they are outside the scope. In EX06 we only care about the data structures to compute the point coordinates inside the C++ program itself; SVG is used just to show the closed paths.)

Suggested Object Model

Please note that there are no restrictions on the implementation as long as you can read the input files and output the transformed shapes as SVG (so you can safely ignore this section).
In the diagram (see Figure 2) we try to implement a few object-orientation patterns and some other tricks. Here is the list:

1. The shapes are built using *Composite Pattern*: `Shape` is an abstract class that materializes as `Group`, `Circle`, `Polygon`; and `Group` itself may contain other shapes, including other groups. Calling methods `draw()` or `transform()` on a `Group` objects, they recursively propagate down the tree: In order to transform a group of shapes (e.g. to rotate some group), we call the same transform on every member of that group.

2. The methods to manipulate shapes are implemented using *Abstract class* and *Pure virtual method* language constructs. The top-level class `Shape` does not know how to draw itself or how to transform itself (consequently, one cannot create objects of this type). Fortunately, the derived classes (`Circle`, `Polygon` or `Group`) consist of concrete points and other shapes, so they can draw/transform themselves.

3. Handling objects of various types by the same code is called *Polymorphism*. For example, iterating over shapes of different kinds in `ShapeStack` is a particular case of polymorphism. Virtual methods ensure that we can call methods like `draw()` and `transform()` on these shapes (and each shape will know, how to apply the method correctly).

4. The transformation matrices are created using *Factory Pattern*: we want to allow creating matrices in just one class: `Transformation`. Creation of 3×3 matrices for affine transformations do not depend on any "state of object", so they can use *Static Methods*. For example, if you want to rotate vector $\mathbf{v} = (x, y) = (3, 5)$ by $72°$ and get another vector $\mathbf{w} = (x', y')$, you can write it like this:

   ```
   Matrix<double> mm = Transformation::getROT(72); // 3x3 matrix
   Matrix<double> vv(3,1); // column vector has 3 rows, 1 column
   vv.a[0][0] = 3; vv.a[1][0] = 5; vv.a[2][0] = 1;
   Matrix<double> ww = mm*vv;
   ```

   This code is cleaner and shorter (compared to calling matrix constructor directly and filling all the $3 \times 3 = 9$ entries manually).

5. In the UML diagram the collection used is *Iterable Stack* – see the class diagram for `ShapeStack`. In addition to all the stack-specific methods (push, pop, etc.) it should also allow to iterate. For example, if we want to apply some affine transformation to all members of some group. Our input handling (and also correct drawing order) depends on the stack behavior. On the other hand "vanilla" stack has very limited set of methods: The stack from your textbook does not support iteration (visiting all elements) easily: You can only look at the top element (and to look deeper you need to pop off the top elements).
   To work around this you can either manipulate two stacks (pop off one of them, push into another one; then do same thing backwards – to restore the correct order). But you can also provide an additional "iterator cursor" variable in the stack that can move around and visit all elements in your stack (without destroying them). Such iterable stacks are sometimes discussed in data structures courses: `https://introcs.cs.princeton.edu/java/43stack/`.

6. Our UML diagram uses some new notation (class inheritance shown as a white triangle; aggregation shown as a white diamond and association shown as simple line). Their differences are explained here - `https://bit.ly/31vjbbJ`.

As usual, the private members (grayed out in the diagram) is something that developers of this diagram can do in whichever way they want; due to encapsulation not even unit tests can test private members. (And you are responsible for reading the STDIN and printing SVG to STDOUT; so let's stress it once more that you should feel free to change this suggested class design any way you want.)

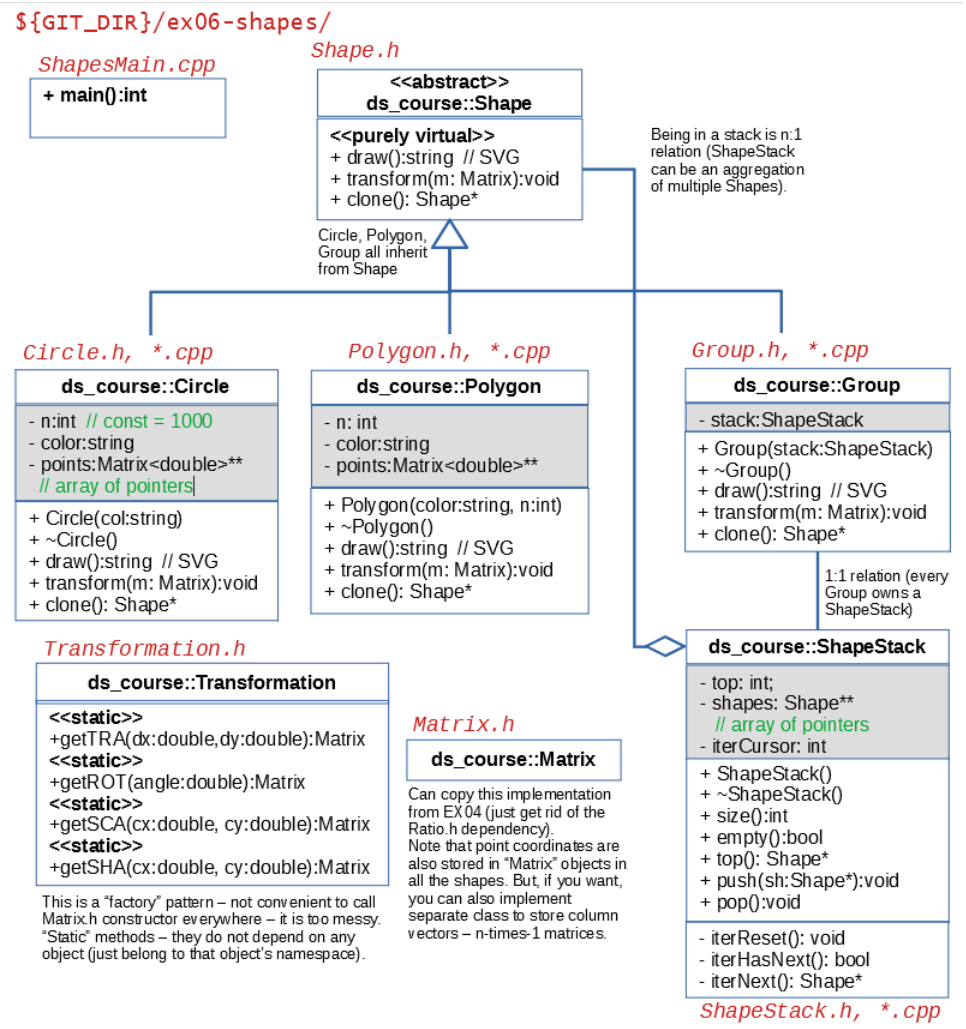


Figure 2: Suggested UML Diagram

Some Pseudocode

Finally, once the object model (with its classes and their interfaces and iterable stack interface with its Abstract Data Type (ADT)) is ready, we can try to show, how this can be put together to parse the input files and to do the requested actions.

```
    MAIN() :
 1  s ← empty ShapeStack
 2  while (getline(cin, Line)) :   (Can read one more line)
 3      cmd ← token from Line  (Read command-name)
 4      if  cmd == "CIRCLE"
 5          color ← token from Line
 6          s.push(new  Circle(color))
 7      if  cmd == "POLYGON"
 8          (color, n) ← 2 tokens from Line
 9          s.push(new  Polygon(color, n))
10      if  cmd == "TRANSF"
11          (type, cx, cy) ← 2 or 3 tokens from Line  (rotation needs just angle)
12          matrix ← Transformation :: getROT(angle)  (or any other transformation matrix)
13          s.top().transform(matrix)  (apply to the stack's top element only)
14      if  cmd == "COPY"
15          newShape ← s.top().clone
16          s.push(newShape)
17      if  cmd == "GROUP"
18          N ← token from Line
19          gStack ← empty ShapeStack
20          repeat N times
21              shp ← s.top()
22              s.pop()
23              gStack.push(shp)
24          s.push(new  Group(gStack))
25      if  cmd == "SHOW"
26          output s.top().draw()   (surround with a few more SVG lines)
27          return 0
```

A sample, how "iterable stack" can be used. Transform a group by transforming each element:

```
    GROUP::TRANSFORM(matrix) :
1   s ← group's instance of ShapeStack
2   s.iterReset()
3   while s.iterHasNext()
4       s.iterNext().transform(matrix)
```