COMP 1633 Assignment 5 (Paint Program)
Parts 1 & 2

Given: **Monday, March 20, 2022**
Electronic Copy Due: **Friday, April 8, 2022 at 11:59 pm sharp**

NO LATES. The course late policy is NOT in effect. Late assignments will not be accepted.

Objectives

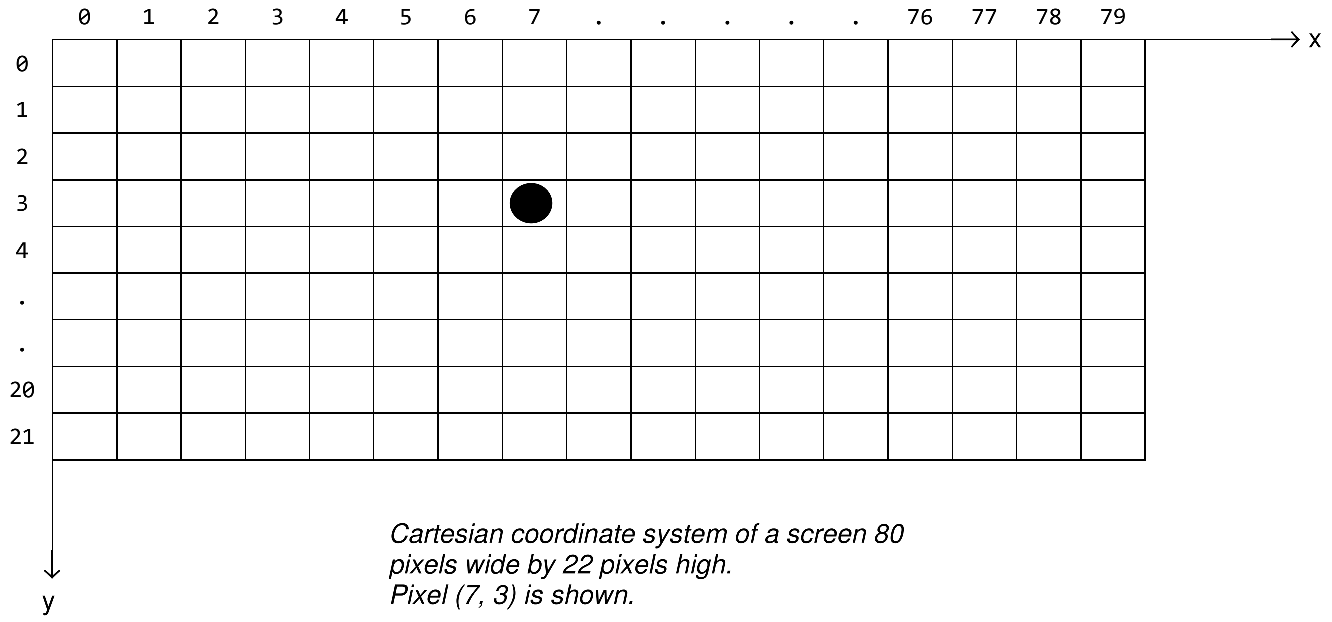
- To build problem solving skills
- To build C++ programming skills
- To gain more experience with classes, objects and object-oriented programming
- To write an application which applies these important concepts
  - overloaded operators
  - managing dynamic resources
  - deep versus shallow copying
  - inheritance and polymorphism
- To practice your software testing skills
- To practice budgeting your time effectively

Background

Almost everyone who has used a computer has, at one time or another used a paint program like Microsoft Paint. Such a program allows the user to draw points, lines and various shapes like rectangles and circles.

Understanding how a paint program works requires a short introduction to raster graphics. A raster is a 2-D grid of pixels (picture elements … points of colour). Here, we only consider monochrome graphics in which a pixel is either ***on*** or ***off***.

In computer graphics, it is conventional to label the upper left corner of the raster as coordinate (0,0). This numbering scheme is compatible with the order in which raster devices such as monitors and printers produce output, although it is upside-down from mathematical convention.



Cartesian coordinate system of a screen 80 pixels wide by 22 pixels high. Pixel (7, 3) is shown.

Monochrome raster graphics can be displayed in a text-only environment. Simply treat a 2-D array of characters as a 2-D grid of pixels. The '*' character represents a pixel in the on state, and ' ' (blank space) represents a pixel in the off state. For example, here is a picture rendered on an `80 × 22` grid:

```
********************************************************************************
*                                                                              *
*                                                                              *
*                                                                              *
*                                                                              *
*                                                                              *
*                                                                              *
*                                   *                                          *
*                                *** ***                                       *
*                             ***       ***                                    *
*                          ***             ***                                 *
*                       ***                   ***                              *
*                      ***************************                             *
*                         *                   *                                *
*                         *  ***         ***  *                                *
*                         *  * *         * *  *                                *
*                         *  ***         ***  *                                *
*                         *        ***        *                                *
*                         *        * *        *                                *
*                         *        * *        *                                *
********************************************************************************
```

The Problem: Overview of All Parts

Your task for this assignment is to write a simple text-based paint program. Ultimately, the program will allow the user to *add*, *delete* and *move* polygons on an `80 x 22` grid. Luckily, polygons are made up of straight lines so you don't need to worry about rendering more complicated shapes such as circles and ovals.

The Parts

Your instructor will grade only your final submission. However, the problem specification will be released in three parts. This handout consists of parts 1 and 2. The rationale is that this problem is relatively complex compared to earlier assignments, and, for success, you need to approach it in a step-by-step, logical way. *Each part builds upon the previous part, which means that components of the previous part will be altered in subsequent parts. However, each part has a testing component that is different from the rest.* ***Therefore, a copy of each complete part MUST be maintained as the final submission will include all three parts****. Thus, when starting a new part you will need to make a copy of the previous part with which you can start working on the new part.*

Project Scheduling

Success in developing larger software systems is based primarily on careful budgeting of available time. To begin acquiring this skill, you should follow these steps:

- Write up a schedule which lists all the tasks you will need to complete to satisfy the specifications for that part.
- Then, estimate the time required for each task.
- Next, sum all the estimates to arrive at the total time estimate for the part and use a daytimer to allocate the work over the available days. Part 3 will be released in about a week.
- Finally, stick to your work plan!

This parallels what real software development teams do in industry, so try to make a habit of planning all of your work in this manner.

Part 1 - The Problem

In this part, your primary goal is to thoroughly test a `Grid` class for displaying monochrome raster graphics. As a side effect, you will gain an understanding of how the `Grid` class does its job. In addition, you will examine a classic computer graphics algorithm - Bresenham's Algorithm - for drawing a line between two points.

The Grid Class

The definition and implementation modules for the `Grid` class are available in the course directory:
`/users/library/comp1633/assignments/a5/p1`

The class provides an `80 × 22` text raster, where `'*'` represents pixel-on and `' '`, a space, represents pixel-off. The public operations are:

a constructor	initializes all pixels to ***off***
`plot_point`	plots a point at coordinates (x,y)
`plot_line`	plots a line segment between (x1,y1) and (x2,y2), inclusive
`write`	writes the raster to the given output stream

The class also supports an overloaded insertion operator<< (aka output operator).

Plotting Points and Lines

Plotting an individual point is straightforward. Drawing a line is a matter of plotting a sequence of points. The sequence of points to be plotted for a horizontal or vertical line is easily calculated given the end points. However, the best algorithm for plotting a slanted line is not obvious. Because the grid is not square, even drawing a diagonal line with slope == 1 is not straightforward.

In 1962, a solution to this problem was devised by Jack Bresenham. He outlined an algorithm for efficiently plotting a line segment joining two endpoints. The `plot_line` method uses Bresenham's algorithm to draw any line.

An explanation of Bresenham's algorithm can be found at
http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Error Handling

It is not an error to attempt to plot a point outside of the grid boundaries. Out-of-bounds points are handled by simply not plotting them. All in-range points must be plotted. For example, the following client code is legal:

```
        grid.plot_line (-100, -100, 100, 100);
```

The above results in a clipped line, i.e. only the segment visible on the grid is actually plotted. Therefore, there is no error handling by the class. A private helper function, `is_in_bounds`, exists to determine whether or not a given point is visible.

Part 1 Deliverables and Milestones

You must test the `Grid` class using `GoogleTest` to perform unit testing of all the public methods of `Grid`.

An initial `test_Grid.cpp` file is provided as a starting point. You have been provided with an initial test case for the default constructor. Notice the inclusion of the `stringstream` and string classes. Carefully observe how the default constructor has been tested – you will need to use this method for your test cases! For purposes of testing, the grid has been reduced to `50 x 15`.

Create a `makefile` that creates the executable for your `GoogleTest` module, runs the test module program and then displays coverage results, using `gcov`.

Part 2 - The Problem

Before you begin this part, you should have a (passing!) `GoogleTest` suite for the `Grid` class, and should be able to verify the functionality and the coverage of the `Grid` class.

Remember to copy your part 1 solution to a new directory for part 2 BEFORE making any changes!!

In this part, you have two goals:

1. Create a `Shape` class that behaves like a square.
2. Complete the implementation of a `Shape_Collection` class that allows a client to maintain a number of shapes.

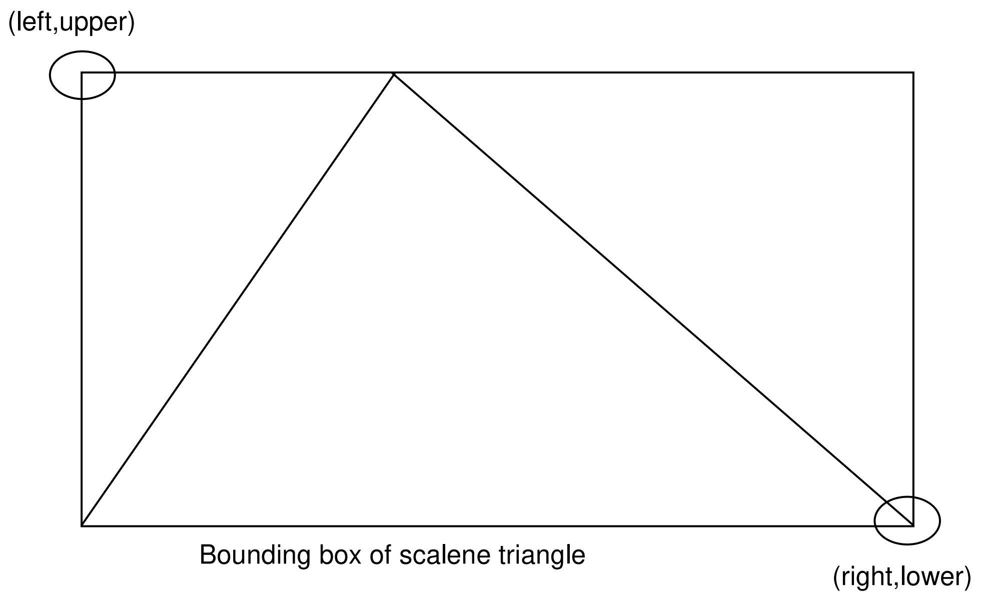
The Shape Class

Background

In graphics, each shape has a bounding box: the smallest rectangle with base parallel to the x-axis which contains that shape. The bounding box is used to specify both the shape's position and its size. The bounding box for a shape can be specified in terms of its (left,upper) and (right,lower) coordinates.

In this assignment, we will only consider polygons with a base parallel to the x-axis. For a rectangle, the bounding box is the rectangle. For a triangle, the width of the bounding box is the width of the triangle's base. The height of the bounding box is the height of the triangle.

For example, here is the bounding box for a scalene triangle:



Public Operations

In this part of the assignment, there is only one kind of shape: a square.

The Shape class must provide the following operations:

- `draw(grid)` *render the shape on a given grid*
- `move(x,y)` *move the shape to position (x,y) where (x,y) specifies the (left,upper) coordinate of the bounding box*

Example Client Code

A client of the class must be able to include code such as the following:

```
Grid aGrid;

Shape s1 (1, 1, 5);                     // square of width 5 at (1,1)
Shape s2 (5, 12, 10);                   // square of width 10 at (5,12)

s1.draw(aGrid);                         // render both squares on the grid
s2.draw(aGrid);

cout << aGrid;                          // OUTPUT #1: write the grid

s1.move(10, 10);                        // move square to (10,10)
s1.draw(aGrid);                         // re-render the moved square

cout << aGrid;                          // OUTPUT #2
```

OUTPUT #1 produces:

```
--------------------------------------------------------------------------------

 *****
 *   *
 *   *
 *   *
 *****






     **********
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     **********
--------------------------------------------------------------------------------
```

OUTPUT #2 produces:

```
--------------------------------------------------------------------------------

 *****
 *   *
 *   *
 *   *
 *****




          *****
          *   *
     **********
     *    *   *
     *    *****
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     **********
--------------------------------------------------------------------------------
```

Note that output #2 is superimposed over output #1. This is because the grid does not get cleared at any time (in fact, it has no clear method). It would be ***possible*** to instantiate more than one grid with the intention of drawing some shapes on one of the grids and other shapes on a different grid, but this will not be done for this assignment.

Error Handling in the Shape Class

A clipped shape is not an error. In other words, a shape may be positioned such that it is only partly visible. In fact, it is possible to instantiate or move a shape so that it is not visible at all!

It would be an error to specify a zero or negative value for a shape's width or height. However, you may assume that the client code will always supply positive numbers for these values.

Therefore, there are no error cases to be handled by the Shape class.

Design and Implementation

The Shape class must be defined as follows:

```
class Shape
{
public:
   Shape(int lft, int up, int width);

   void move(int new_left, int new_upper);

   void draw(Grid& grid);

protected:
   void get_bounding_box(int& lft, int& up, int& rght, int& low) const;

private:
   int left;
   int upper;
   int right;
   int lower;
};
```

The constructor initializes the bounding box information for the shape being created. A shape's bounding box cannot be queried by client code. The protected helper method `get_bounding_box` is available only to other member functions.

The `move` operation requires the new (left,upper) coordinate of the shape's bounding box.

The Shape Collection Class

High Level Description

Ultimately, the paint program will allow the user to create and manipulate several shapes at a time. A given picture might consist of 0, 1, 25, or even 250 shapes! This means that the program must maintain a dynamic collection of Shape objects. Further, the user needs a mechanism to select a particular shape and then move it or even remove it from the picture. This means that each shape must be identifiable. We will associate a name of the user's choosing with each new shape that is added to the picture. A name is a sequence of printable characters that does not have any leading, trailing or internal whitespace characters.

The order of the elements contained in the collection is irrelevant.

Public Operations

The `Shape_Collection` class must provide the following operations:

• `add(shape, name)`	*add the given shape with a given name to the collection*
• `remove(name)`	*remove the shape with the given name from the collection*
• `lookup(name)`	return a pointer to *the shape with the given name*
• `draw_shapes(out)`	*draw all shapes in the collection to the given* ostream
• `write_names(out)`	*write the names of all shapes to the given* ostream

In fact, when adding a shape to the collection, the client will pass a pointer to a dynamically allocated object. The collection will then take custody of the object, and will ultimately be responsible for deallocating it.

Error Handling in the Shape Class

For simplicity, you may assume that the client will not add duplicate shape pointers (i.e. a pointer to a `Shape` object already in the `Shape_Collection` object) nor duplicate names to the collection.

The `remove` method does nothing if a shape with the given name does not exist. The `lookup` method returns `NULL` if a shape with the given name does not exist.

It is not an error to invoke `draw_shapes` or `write_names` on an empty collection.

Example Client Code

A client of the `Shape_Collection` class must be able to include code such as the following:

```
Shape_Collection myShapes;
Shape* aShape;

aShape = new Shape(1, 1, 5);
myShapes.add(aShape, "s1");

aShape = new Shape(5, 12, 10);
myShapes.add(aShape, "s2");

myShapes.write_names(cout);     // OUTPUT SECTION #1
cout << endl;
myShapes.draw_shapes(cout);

aShape = myShapes.lookup("s1");

if (aShape != NULL)
   aShape->move(10, 10);
myShapes.remove("s2");

myShapes.write_names(cout);     // OUTPUT SECTION #2
cout << endl;
myShapes.draw_shapes(cout);
```

Output Section #1 produces:

```
[s2, s1]
--------------------------------------------------------------------------------

 *****
 *   *
 *   *
 *   *
 *****






     **********
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     *        *
     **********
--------------------------------------------------------------------------------
```

Output Section #2 produces:

```
[s1]
--------------------------------------------------------------------------------










          *****
          *   *
          *   *
          *   *
          *****







--------------------------------------------------------------------------------
```

Note the Shape names when printed, before the Grid, are in reverse order since each shape is added to the front of the collection. You will also notice that previously drawn shapes do not show up when the grid is drawn. This is because there is no permanent `Grid` object instantiated and used - each time the `draw_shapes` method is invoked, it will instantiate a grid, draw on that grid and then display it. When the method returns, the `Grid` object goes out of scope.

Design and Implementation

An incomplete `Shape_Collection` class is provided. Copy `Shape_Collection.h` and `Shape_Collection.cpp` to your working directory from:

```
/users/library/comp1633/assignments/a5/p2
```

A collection is represented internally as a doubly linked list of shapes, each with an associated name. Note that we are using the C++ standard string class instead of C-strings. Please refer to Section 8.2 of your textbook for additional information.

Most of the member functions are already complete! You must finish the implementation for the following two methods:

- `draw_shapes`
- `write_names`

You should be able to pick some hints and tips by studying the implementation of the other methods.

Deep Copying

Because the `Shape_Collection` class manages dynamic resources, technically it requires a deep copy constructor and assignment operator to override the ones provided as the system defaults. However, these operations are difficult to write correctly for this class (try it if you like). Therefore, you are not required to write deep copy methods for this assignment.

However, it would be dangerous to leave the client with the ability to perform *buggy* shallow copying! In situations such as this one, the class implementer can provide an overridden copy constructor and assignment operator in the private section of the class definition and then simply stub out their implementations. This means that the client will never be able to perform copying or assigning of class objects (including passing or returning objects by value).

For example, by making these private stubs, the following will not compile:

```
      Shape_Collection shapes;
      Shape_Collection shapes2;
      .
      .
      .
      Shape_Collection dup(shapes);     // COMPILE ERROR HERE!  NOT ALLOWED
      .
      .
      .
      shapes2 = shapes;                 // COMPILE ERROR HERE!  NOT ALLOWED
```

Part 2 Deliverables and Milestones:

For this part of the assignment, you need to do the following:

1. Create the definition module (see page 8) and develop the implementation module for the `Shape` class.

2. Finish the implementation and documentation of the `draw_shapes` and `write_names` methods of the `Shape_Collection` class.

3. Write a collection of unit tests using `GoogleTest` to ensure that the Shape and `Shape_Collection` classes are working properly. Every public method must be appropriately tested.

4. Add appropriate targets and rules to your `makefile` so that these new components of the final project can be built. The testing of both components must include rules that demonstrate coverage of your code using `gcov`.

Submission and Grading

You will not be submitting Parts 1 and 2 independently of the final project (Part 3). However, you should aim to have it completed by approximately March 31st so that you can start on Part 3 once it is released. At that point, there will be just over one week left to complete the assignment!

Coding Style and Documentation

Write well structured, readable and self-documenting code. Concentrate on high quality documentation which explains the code clearly, while remaining concise. As always, follow the course coding and documentation standards. Poor style and low quality documentation will be graded harshly.

Each library module should have a brief identification block that contains the author, date and file name at a minimum. Each library header module should contain a block describing each public member function, while each implementation module should have a block describing the implementation method for the class and a description for each private function. The descriptions should follow the examples given in tutorials.

It is suggested that you document your functions as you implement them (or better yet, *before* you implement them). There is no need to document code that you don't create.

Submission

Submission instructions will be given on Tuesday, April 5th.