Graded Assignment 3: Drawing on the Terminal

Due date: May 27, 2018 at 20:00

This is an individual assignment. You may discuss it with others, but your formulations, your code, and all the required material must be written on your own. In any case, you must acknowledge the sources used and clearly mention any help received from colleagues.

1 Problem

You must implement a graphic library that works in the terminal and that can be used to draw simple geometric shapes using colored characters as pixels.

The library is based on a Cartesian coordinate system, with a horizontal x-axis and a vertical y-axis starting from the origin at the top left corner of the terminal window, and extending to the right bottom corner. Thus the coordinates of the top left corner are $(0, 0)$ and the intersection of the line in position y from the top and the column in position x from the left contain the character in position (x, y). The exact number of lines and columns in a terminal window may vary.

2 Implementation

Your main tool for drawing in the terminal window is the `Screen` class provided in `include/screen.h` header file. A `Screen` object represents a buffer of characters, one character for each position in the terminal window. An application would use a `Screen` object to draw shapes on the screen.

2.1 Part 1

The `include/screen.h` and `src/screen.cpp` source files already define and implement some methods. You have to implement other methods as specified below.

There are two main classes `Point` and `Screen`, the `Point` class represent a "pixel" in the terminal, it has a character, a code for the foreground and background colors, and a flag that indicates whether the pixel is set to be "bright". Most terminals use ANSI escape sequences (sequences of bytes) that instruct the terminal to activate various colors. You must use these ANSI escape sequences to control the terminal.[1]

In `Screen` you can draw primitive geometric shapes, in this case, the circle shape is already implemented in `src/screen.cpp`. Your task is to implement the functions marked as "TODO" in the source files, which are specified as follows:

- `~Screen()`: destructor for the `Screen` class.
- `render()`: render output into the terminal.
- `init()`: initialize the screen buffer.
- `void set_rect(int x0, int y0, int x1, int y1, const Point & p)`: draw a rectangle. Notice that the actual output is done by `render()`. The top left corner of the rectangle is at coordinates $(x0, y0)$, and the bottom-right corner is at coordinates $(x1, y1)$. The `p` parameter is a `Point` object, passed by reference, that defines the properties of the rectangle.

[1] See the ANSI color codes as defined on Wikipedia: `https://en.wikipedia.org/wiki/ANSI_escape_code#Colors`.

- `void set_line(int x0, int y0, int x1, int y1, const Point & p)`: Similar to `set_rect`, draws a line from $(x0, y0)$ to $(y0, y1)$. Only horizontal, vertical, or 45-degree lines are permitted.

The other functions of `Screen` are as follows:

- `int ncols() const`: horizontal size of the terminal window, that is, the number of columns.
- `int nrows() const`: vertical size of the terminal window, that is, the number of rows.
- `void clear()`: cleans the terminal window and sets cursor to the top left corner. This is useful to clean the screen before drawing shapes.
- `void set_circle(int x, int y, int r, const Point & p)`: similarly to `set_rect`, draws a circle centered at (x, y) of radius r.

2.2 Part 2

As you can see, the `Screen` class provides a low-level interface and therefore may be difficult to use. For example, one has to memorize color codes. You are therefore required to also implement the following utility classes:

- `Color`: a class serving as a wrapper around a plain color code (an integer). There should be predefined colors available through constant static fields and construction of new colors should be prohibited to the clients. Colors codes are also described on-line (see the Wikipedia page referenced above). You should implement the following eight colors, both for foreground and background:
  - `color::black`
  - `color::red`
  - `color::green`
  - `color::yellow`
  - `color::blue`
  - `color::magenta`
  - `color::cyan`
  - `color::white`
- `Pen`: a class containing information about a character to use for drawing, as well as its foreground and background colors and brightness. Colors are of integral type. Brightness is a Boolean value, since a "pixel" can either be bright or not bright. You must implement the following methods:
  - Constructor `Pen(int fg_color, int bg_color, char c, bool brightness)`
- `Shape`: an abstract class representing various shapes and serving as a polymorphic base for other specific classes. Since it is a virtual base, `Shape` has a virtual destructor. `Shape` also has a pure virtual method `void draw(Screen & screen, const Pen & p) const` that uses the functionality of the given screen to draw itself (not render) using pen p.
- `Rectangle`: a concrete class, derived from `Shape`, that represents a rectangle. `Rectangle` keeps information about the top-left corner of the rectangle as well as the width and height of the rectangle, which is usually better than keeping track of two opposite corners of the rectangle since most calculations are more concise when we know the width and the height. The state of the rectangle is set once during construction, but it can be read later using public access methods ("getters"), one per each private field. Finally, a rectangle, being derived from `shape` must implement the virtual method `draw` to actually draw a rectangle. In summary, you must implement the following methods:
  - Constructor `Rectangle(int x, int y, int width, int height)`.

  - `int get_width() const`: return the width of the rectangle.
  - `int get_height() const`: return the height of the rectangle.
  - `int get_x() const`: return the x coordinate of the top left corner of the rectangle.
  - `int get_y() const`: return the y coordinate of the top left corner of the rectangle

- `Circle`: a concrete class derived from `Shape` representing a circle. It is similar to rectangle, except that a circle is defined by three fields: two integer coordinates of the center, and the radius. You must also implement the `draw` methods. The constructor is as follows:
  - `Circle(int x, int y, int radius)`

- `Line`: a concrete class derived from shape representing a line segment. Similar to rectangle, a line contains the coordinates of the two points that define the segment. The constructor, `Line(int x0, int y0, int x1, int y1)`, must check that the line segment is axis-aligned or at 45 degrees, and otherwise throw an integer value 1 as an exception. You must also implement that `draw` method. implement the functions:

- `Canvas`: a class representing an abstraction over the screen. `Canvas` contains a reference to an existing instance of class screen, and a list of *Shape–Pen* pairs $(s_1, p_1), (s_2, p_2), \ldots$ that specifies that shape s_1 must be drawn with pen p_1, s_2 must be drawn with pen p_2, etc. (*Hint:* the standard library provides various container classes, as well as a `pair` utility class.) In summary, you must implement the following methods:
  - Constructor `Canvas(Screen & screen)`.
  - `add(Shape * shape, Pen & pen)`: add a pair (shape, pen) to the canvas list.
  - `clear()`: clear the list of shapes as well as the screen
  - `show()`: display the shapes from the list in the terminal window.

You must implement all these classes and their methods in the provided files `include/cscreen.h` and `src/cscreen.cpp`.

3 Test Cases Plus Extra Exercise

Along with this document, we provide some test cases you might run to test your implementation. A test failure indicates that your code is not implemented according to the specification.

The `./tests/test.sh` script tests your implementation with all the conformance tests. Make sure your program passes the tests before submitting. Notice that the tests in `./tests/` require a window size of 80×40. However, your solution should work, and will be tested, against several terminal sizes.

3.1 Part 1

Test case `src/example1_2.cpp` must print something similar to a student that succeeded this assignment, in the likes of Figure 1.

3.2 Part 2

The test in `./src/example2.cpp` uses your library to produce something similar to a frightened robot, as depicted in Figure 2. Notice that `./src/example2.cpp` has a memory leak. You must fix the code and return it back with your submission.

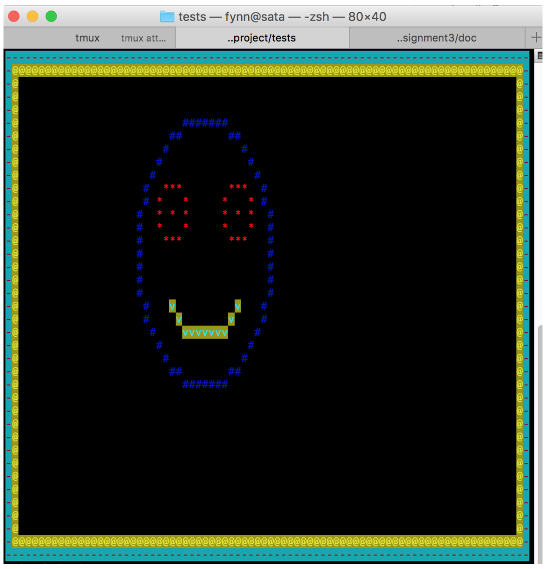

Figure 1: Example of output for part 1

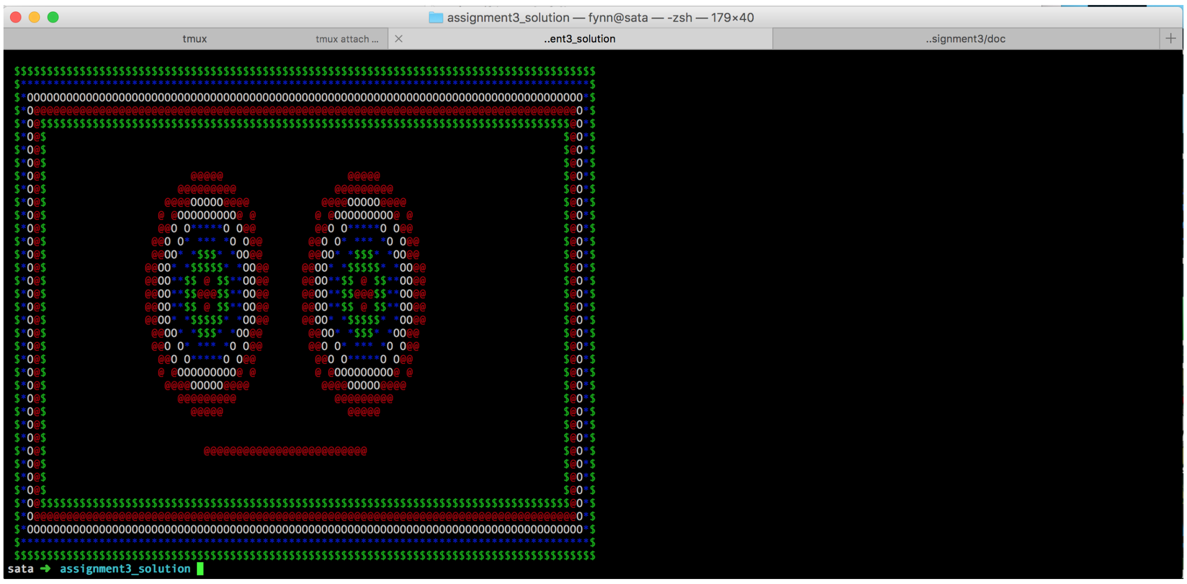
assignment3_solution — fynn@sata — -zsh — 179×40
tmux
tmux attach ...
..ent3_solution
..signment3/doc
sata ➜ assignment3_solution

Figure 2: Example of output for part 2

4 Testing and Submitting Your Solution

You are provided with a `Makefile` to automate the build and testing process. You may just run `make` in the source directory to build the library. You can also compile each test individually with `make example1_1`, `make example1_2` and `make example2`, respectively. To run the tests, compile the code and then enter the `tests/` directory and run `./test.sh`.

You must submit a package (tar.gz or zip) containing all the necessary pieces to build and test if your solution conforms to the specification. You can produce a submission package using the provided Makefile, by running `make submission`. This will create a file called `submission.tar.gz`. Submit that file on-line through the iCorsi system.

Don't forget to fix the memory leak in `src/example2.cpp`!