Προγραμματιστικές Τεχνικές: Εργασία 6

Προθεσμία: Παρασκευή, 20 Ιανουαρίου 2017

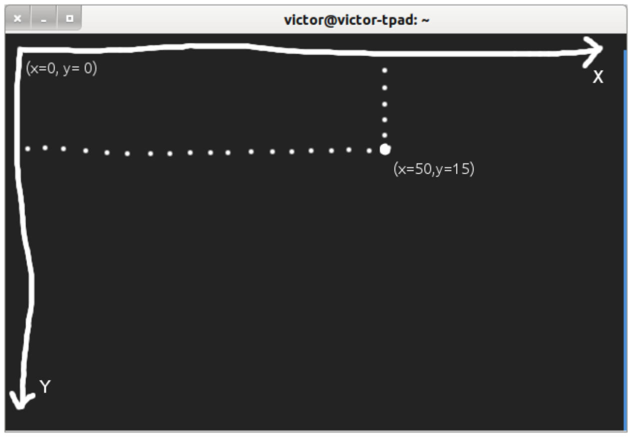
Designing Object-Oriented Systems in C++

This assignment is dedicated to designing a small object-oriented system for generating pseudo-graphics in the terminal's window. This assignment covers basics of object-oriented design, polymorphism, abstract classes and handling pointers/references.

`Graphic Primitives as Objects`

When people start learning object-oriented programming, they inadvertently encounter example classes representing shapes, such as rectangles or circles. We are going to implement a few such classes. However, in order not to bother with setting up and learning any advanced graphics libraries, such as OpenGL, you are provided with a small graphics subsystem `ps6.h` that allows "drawing" a few simple geometric pseudo-shapes in the terminal's window, using colored characters as "pixels".

Prior to describing how to use `screen`, we agree on the coordinate system: it is Cartesian, with horizontal `x` axis and vertical `y` axis pointing from the origin at the top left corner of the terminal's window. All coordinates are integer.



Example coordinates

The top left corner's coordinates are `(0, 0)`. A single unit along each axis is of the size of a single character, i.e., characters are our "pixels". Thus, the size of the terminal window is measured in the number of lines (y axis) and the number of columns (x axis), where intersection of a line and a column contains a character. The exact number of lines and columns in a terminal window vary (by default, the terminal window' size may be around 80 columns by 40 rows, but it may be different on your system, and does not matter anyway).

Your main tool for drawing in the terminal's window is `class screen` provided in `ps6.h` (it is both declared and defined there, just to keep everything in one file). First of all, screen is an instantiable class, that is, it is not stateless, and, in order to use it, one needs to create an object of type `screen`. It requires the size of the screen to use in order to be constructed, i.e., `scr(cols, rows)`.

`screen` represents a buffer of characters, one character for each position in the terminal's window. We can ask `screen` to "draw" some shape on the screen. In response, it will just update its buffer. This way, we can submit multiple requests to `screen` and, then, call its method `render()`, which will actually produce the characters from the buffer to the terminal's window. Here is a short summary of `screen`'s methods:

- `int ncols() const` — returns the horizontal size of terminal's window (number of columns)
- `int nrows() const` — returns the vertical size of the terminal's window (number of rows)
- `void clear() const` — cleans the terminal's window and sets cursor to the top left corner (useful to clean the screen before drawing shapes)
- `void set_rect(int x0, int y0, int x1, int y1, const point &pt)` — draws a rectangle (as said above, it draws it in the buffer; actual output is done by `render()`). `(x0, y0)` are the coordinates of the top left corner of the rectangle, `(x1, y1)` are the coordinates of its bottom right corner. `pt` is a structure that defines which character will be used for drawing as well as its colors and brightness (see `ps6.h` to figure out how `struct point` is defined).
- `void set_circle(int x, int y, int rad, const point &pt)` — similarly to `set_rect`, "draws" a circle centered at `(x, y)` of radius `rad`.
- `void render()` — actually outputs everything "drawn" to the terminal's window.

And here is a short example of how one can use `screen`:

```
g++ -g3 -O0 -Wall screen-example.cpp -o example
```

```
#include <iostream>
#include "ps6.h"

int main() {
  using namespace tui;

  screen scr(80,30);
  scr.clear();

  // color codes are defined here; 9 is the code for the default color
  point ptred('@', 1, 3, false);
  point ptblue('#', 4, 0, true);

  scr.set_rect(0, 0, scr.ncols() - 1, scr.nrows() - 1, ptred);
  scr.set_circle(30, 15, 10, ptblue);

  scr.render();

  return 0;
}
```

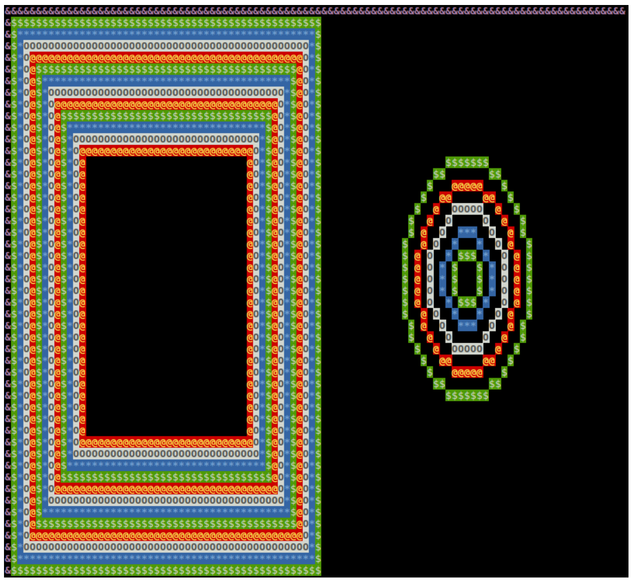
As you can see, `screen` is not very nice (e.g., it uses some cryptic color codes and it is not very object-oriented overall). Thus, your goal is to write a few classes that would hide ugliness of `screen` and provide a neat interface for drawing in terminal. You are expected to implement the following classes:

- `color` — a class serving as a wrapper around a plain (integer) color code. There should be a few predefined colors available through constant static fields and construction of new colors should be prohibited to the clients (the only constructor of class `color` should be private). Colors codes are described in the first small color table here. The goal is to (pre)define a few colors, so that clients could use them as `color::red`.
- `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 type `color`. Brightness is boolean — a "pixel" is either bright or not.

- `shape` — an abstract class standing for some shape. Serving as a polymorphic base class, it contains a virtual destructor. Additionally, it contains a pure virtual method `void draw(screen &scr, const pen &p) const` that uses functionality of `scr` to draw itself (not render) using pen `p`.
- `rectangle` — a concrete class derived from `shape` representing a rectangle. It keeps information about the top left corner of the rectangle as well as the rectangle's width and height (it is usually better than keeping track of two opposite corners of a rectangle, since most calculations get more concise when we know width and height). The state of the rectangle is set once during construction, but it can be read later using public getters (one per each private field). Finally, our `rectangle`, being derived from `shape`, implements the pure virtual `draw` to actually draw a rectangle (base class does not know how to draw).
- `circle` — a concrete class derived from `shape` representing a circle. It is similar to `rectangle`, except a circle's state contains three fields — two integer coordinates of the circle's center and the radius. `circle` also implements `draw`, yet, it clearly does it differently from `rectangle`.
- `canvas` — a class representing an abstraction over the `screen`. Contains a reference to an existing instance of `class screen` and a list of (shape, pen) pairs, where each shape is drawn using the pen from its pair. This is where you may need to use templates. The easiest way to store a list of pairs is to use a `vector< pair<Type1, Type2> >`. Both `vector<T>` and `pair<T1,T2>` are members of the Standard C++ Library. You can look up how to use them. Canvas is constructed given a reference to a screen object, and it contains three methods: `add` for adding a pair (shape, pen) on the list, `clear` for cleaning the list of shapes as well as the screen, and `show` for displaying the shapes from the list in the terminal's window. One of the most important things to understand here is how to store a list of objects, so that we can use them polymorphically. We cannot store a list of shape objects, because polymorphism does not work through objects. We need to store pointers to shapes. Pens, however, are not polymorphic, so they can be stored as objects.

Declarations of your classes should be put in `main.cpp`, along with the implementations. All your classes should live in `namespace tui` like `class screen` does.

To better understand how your system should be able to function, take a look at the code in `main.cpp`. You implementation should be such that this code compiles and produces something close to the following (the picture may vary depending on the size of your terminal's window) when given as inputs `100 50`:



Example output for inputs `100 50`