Week9 : Graphics Classes

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 9

Outline

Outline

Overview of the graphics class interface

- GUI libraries tend to have hundreds of classes, each with dozens of methods
- With such large amounts of code it can be a bit hard to figure out where to start
- We will use interface library of B.Stroustrup that is meant to reduce that to about two dozen classes
- This reduced set of classes should make it easier to produce useful displays
- Also it will allow the key concepts involved in using a typical GUI library to be understood
- Using the concepts from these classes you could then extend to using other graphics libraries

Summary of graphics interface classes (1/2)

Graphics interface classes (1/2)

`Color`	used for shapes, and fill
`Line_style`	used for drawing lines
`Point`	x, y coordinates within `Window`
`Line`	line defined by two `Points`
`Open_polyline`	sequence of `Points` connected by lines
`Closed_polyline`	`Open_polyline` but connect last to first
`Polygon`	`Closed_polyline` with no intersecting lines
`Text`	string of characters at x, y
`Lines`	set of line segments by pairs of `Point`S

Summary of graphics interface classes (2/2)

Graphics interface classes

`Rectangle`	commonly used shape for displays
`Circle`	a circle is defined by center and radius
`Ellipse`	an ellipse has a center and to axis lengths
`Function`	a function of one variable in a range
`Axis`	a labelled axis
`Mark`	a point marked by a character (eg. '+')
`Marks`	a sequence of points indicated by marks
`Marked_polyline`	an `Open_polyline`, marks on points
`Image`	image from a file

Summary of GUI interface classes

While graphics interface lets us display things on the window, the GUI interface lets us make the window and get input from the user.

GUI interface classes

`Window`	an area of the screen in which we display
`Simple_window`	a window with a "Next" button
`Button`	a rectangle with a label – we click on it to run one of our functions
`In_box`	a box with label in which user types string
`Out_box`	a box with label in which program writes string
`Menu`	a vector of `Buttons`

Graphics interface source files

The source code is organized into the files:

Graphics interface source files

`Point.h`	`Point`
`Graph.h`	all other graphics classes
`Window.h`	`Window`
`Simple_window.h`	`Simple_window`
`GUI.h`	`Button` and other GUI classes
`Graph.cpp`	implement interfaces of Graph.h
`Window.cpp`	implement interfaces of Window.h
`GUI.cpp`	implement interfaces of GUI.h

In addition a container for `Shape` or `Widget` objects is defined as `Vector_ref`, which is a `vector` which can hold un-named elements

Purpose of this lecture

- See correspondence between code and pictures produced
- Get used to reading code and how it works
- Thinking about design of code

Point

- Most basic part of any graphics system is a point
- Typical computer-screen is two-dimensional, so define points by integer (x, y) coordinates
- Defined in `Point.h`, a `Point` is a pair of `int` values:

```
1 struct Point {
2   int x, y;
3 };
4 bool operator==(Point a, Point b){
5   return a.x==b.x && a.y==b.y ;
6 }
7 bool operator!=(Point a, Point b){
8   return !( a == b );
9 }
```

Line

- A `Line` is a kind of `Shape`, and as defined in `Graph.h`:

```
1 struct Line : Shape {
2   Line( Point p1, Point p2 );
3 };
```

- Above definition says that `Line` inherits from `Shape` – that is what the `:` means.
- `Shape` is called a base class for `Line`
- `Shape` provides facilities needed to make the definition of `Line` easier
- `Line` is defined by two points
- We will see more how inheritance can help us next class

Line

- We can create and draw `Lines` as follows:

lineTests.cpp

```
1   // Draw two lines
2   Point winloc { 100, 100 };
3   Simple_window win{ winloc, 600, 400, "Two
        lines"};
4   Line positive{ Point{100,300}, Point{500,100} };
5   Line negative{ Point{100,100}, Point{500,300} };
6   positive.set_color( Color::red );
7   negative.set_color( Color::blue );
8   win.attach( positive );
9   win.attach( negative );
10  win.wait_for_button();
```

Results of lineTests.cpp

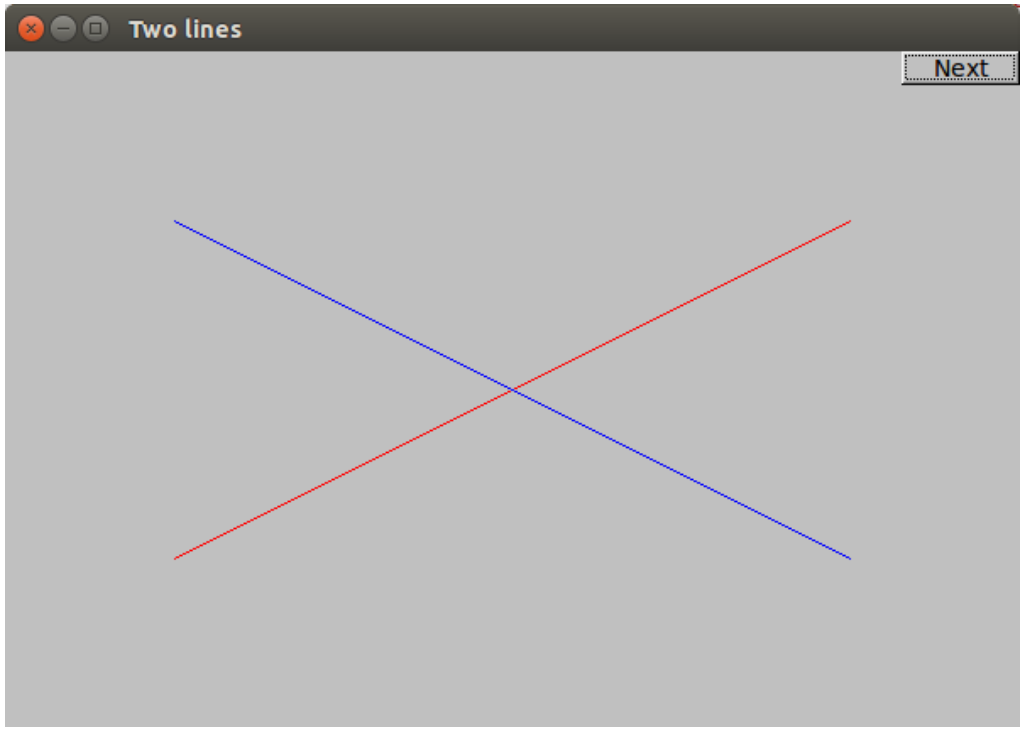

Figure : Result of line drawing test

Lines

- Often we need to draw more than one line
- `Lines` is a collection of lines, defined by pairs of `Points`

```
1 struct Lines : Shape {
2   Lines() {} // empty set of lines
3   // initialize from initializer list
4   Lines( initializer_list< pair<Point, Point> >
        lst );
5   void draw_lines() const;
6   // add a line defined by two points
7   void add( Point p1, Point p2 );
8 };
```

Example `Lines`

- Using `Lines` instead of `Line` implies that the lines somehow belong together.
- If we have several `Line` objects we can change their color individually – but a `Lines` object will have all its lines the same color
- Lets draw two lines in the shape of a large +

```
1 Lines plus;
2 // add a horizontal line
3 plus.add( Point{100,200}, Point{500,200});
4 // add a vertical line
5 plus.add(Point{300,100}, Point{300,300});
```

Results of adding `Lines` to lineTests.cpp

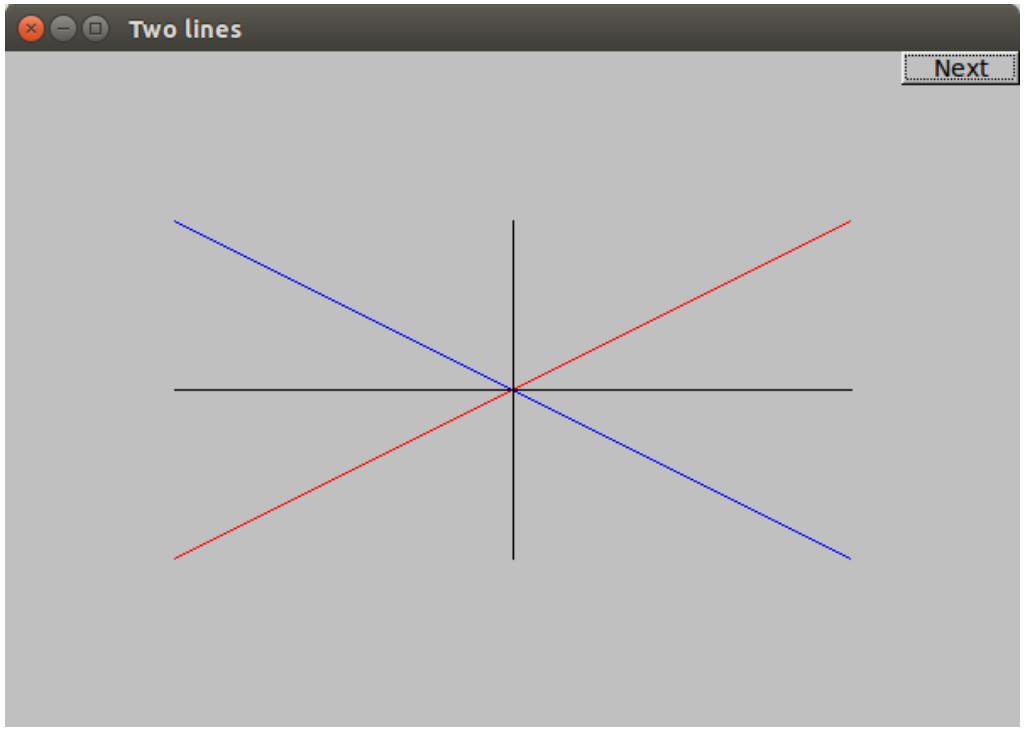

Figure : Result of both line drawing tests

Use of `Lines` to make a grid

- Lets make a grid of lines to divide our window into some number of divisions

drawGrid.cpp

```
1   Simple_window win{ winloc, 600, 400, "grid"};
2   // get back the window size
3   int ww = win.x_max();
4   int hh = win.y_max();
5   const int ntickx = 10; // vert. lines
6   const int nticky =  8; // hor. lines
7   int dx = ww / ntickx; // dist betw vert lines
8   int dy = hh / nticky; // dist betw hor. lines
9   Lines grid;
10  for ( int x = dx; x < ww; x += dx ){
11    grid.add( Point{ x, 0 }, Point{ x, hh } );
12  }
13  for ( int y = dy; y < hh; y += dy ){
14    grid.add( Point{ 0, y }, Point{ ww, y } );
15  }
16  win.attach( grid );
```

Result of running `drawGrid.exe`

- Note how we have done a computation to figure out where to put the grid lines
- That saved us from drawing each line individually

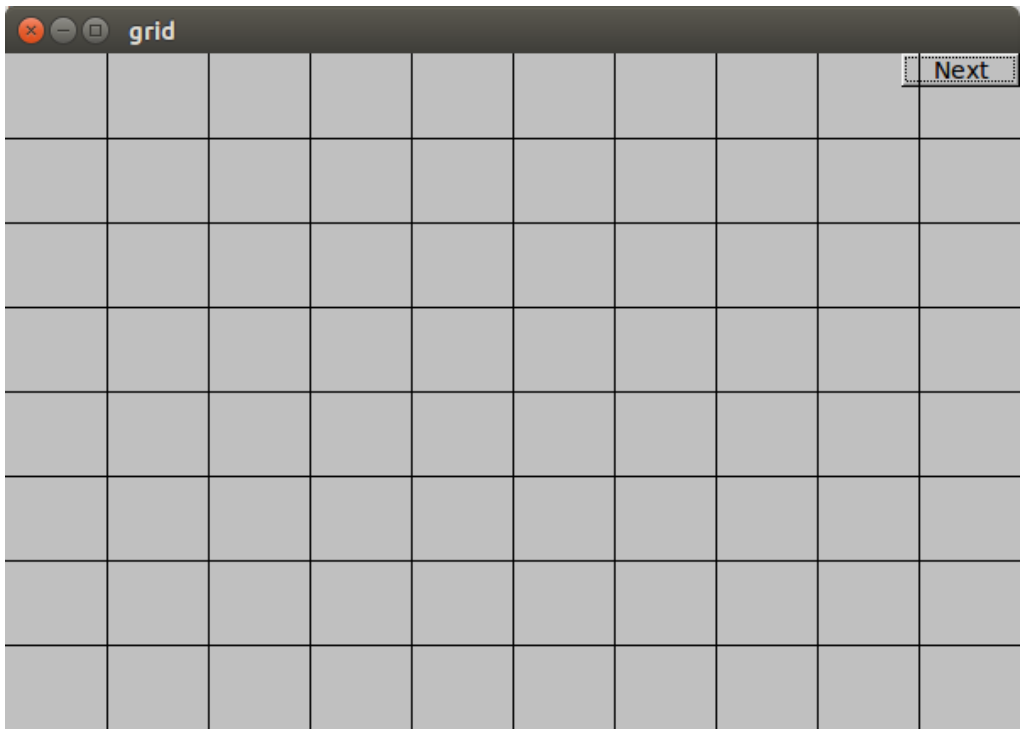

Figure : Result of drawGrid.cpp

Notes on `Lines::add` implementation

- The `Lines` class provides two constructors, and a method `add` to add a line by a pair of points.
- Consider the add method of the `Lines` class:

```
1 void Lines::add( Point p1, Point p2 ){
2   Shape::add( p1 );
3   Shape::add( p2 );
4 }
```

- Note that `Lines::add` adds the points to the underlying Shape by calling `Shape::add`
- Just calling `add(p1)` from within `Lines::add` would result in a compiler error as it see `Lines` own add method, rather than the one in `Shape`

Notes on `Lines::draw_lines()` implementation

- The `draw_lines` method of `Lines` draws the lines defined using `add()`

```
1 void Lines::draw_lines() const {
2   if ( color().visibility() ) {
3     for ( int i=1; i< number_of_points(); i += 2 ){
4       fl_line( point(i-1).x , point(i-1).y,
              point(i).x, point(i).y );
5     }
6   }
7 }
```

- For loop uses two points at a time starting with 0 and 1
- Draws line between them using underlying library's drawing function (`fl_line()`)
- Visibility is property of `Lines' Color` object.
- Can only add pairs of points, so no need to check if `number_of_points()` is even
- `number_of_points()` and `point()` are members of the `Shape` class, and have obvious meaning
- Note that `draw_lines()` is `const` since it doesn't modify the `Lines` object.

`initializer_list` constructor of `Lines`

- By having `initializer_list` constructor that is list of pairs of points, we can build a new `Lines` object using the notation:

```
// make a horizontal and a vertical line
Lines mylines = {
  { Point{100, 100}, Point{200, 100} },
  { Point{150, 50}, Point{150, 150} } };
// or even like this:
Lines cross = {
  { {100,300}, {500,100} },
  { {100,100}, {500,300} };
```

- The initializer-list constructor is defined as:

```
void Lines::Lines( initializer_list< pair<Point,
    Point> > lst ){
  for ( auto p : lst ){
    add( p.first, p.second );
}
```

std::pair

- `std::pair` is a container in standard library that can hold two objects of the same type
- It has two members, named `first` and `second` constructor that are used to get the first and second of the objects making up the pair
- to make a pair of integers: `pair <int, int > p 5, 4`
- then `p.first` returns 5, and `p.second` returns 4
- to make a pair of Points:

```
pair <Point, Point> pp{ Point{100,100},
    Point{200,200} }
```

- Then `pp.first` gives the first point, and `pp.second` gives the second point
- In our for loop, `auto` is a `pair<Point,Point>`

Changing color of `Shape` objects

- We can set a `Shape`'s Color, such as for our grid: `grid.set_color(Color::red)`
- Colors available are in the interface to `Color`:

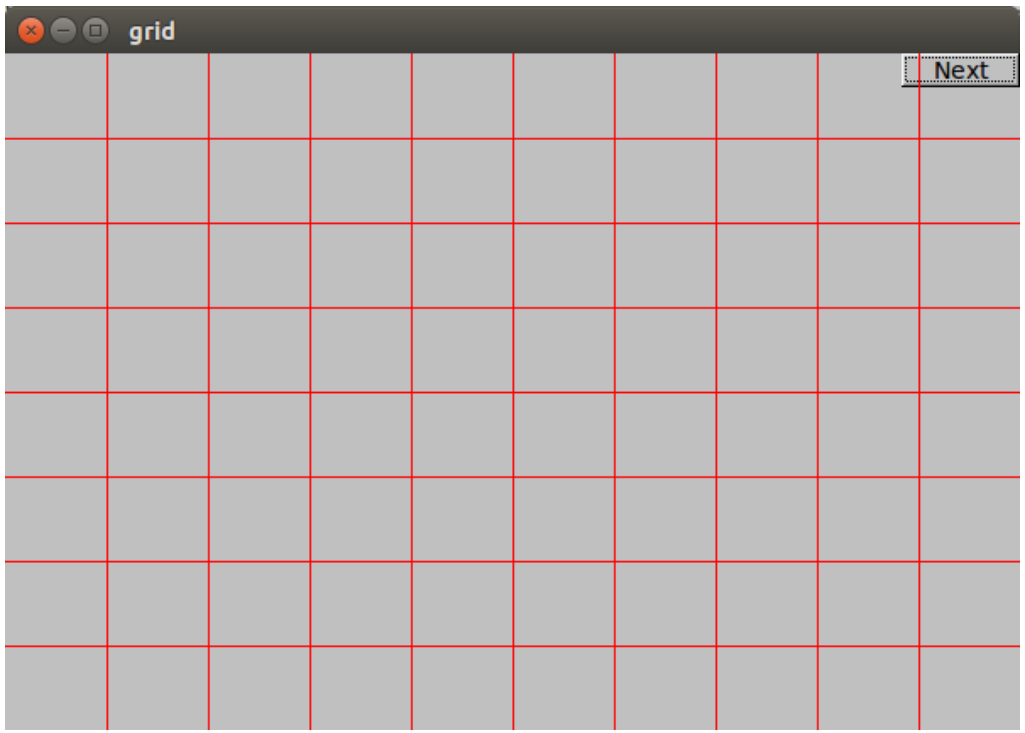

Figure : Turning the grid red

`Color` to represent color and map to `fltk` colors

```
1   struct Color {
2     enum Color_type {
3       red            = FL_RED,
4       blue           = FL_BLUE,
5       green          = FL_GREEN,
6       yellow         = FL_YELLOW,
7       white          = FL_WHITE,
8       black          = FL_BLACK,
9       magenta        = FL_MAGENTA,
10      cyan           = FL_CYAN,
11      dark_red       = FL_DARK_RED,
12      dark_green     = FL_DARK_GREEN,
13      dark_yellow    = FL_DARK_YELLOW,
14      dark_blue      = FL_DARK_BLUE,
15      dark_magenta = FL_DARK_MAGENTA,
16      dark_cyan      = FL_DARK_CYAN
17    };
18    // ... continued on next slide
19  }
```

Color continued ...

```
1  struct Color {
2    // ... continued
3      enum Transparency { invisible = 0, visible=255 };
4      Color( Color_type cc ) :
5           c( Fl_Color(cc) ), v( visible ) { }
6      Color( Color_type cc, Transparency vv) :
7           c( Fl_Color(cc) ), v( vv) { }
8      Color( int cc ) :
9           c( Fl_Color(cc) ), v( visible ) { }
10     Color(Transparency vv) :
11          c( Fl_Color() ), v( vv ) { }
12     int as_int() const { return c; }
13     char visibility() const { return v; }
14     void set_visibility(Transparency vv) { v=vv; }
15   private:
16     Fl_Color c;
17     unsigned char v; // 0 or 255 for now
18   };
```

Notes on `Color`

- Hides FLTK's `Fl_Color` type, and provides a map of those colors
- Gives the color enum a scope
- Provides a method to make `Shape` visible or invisible
- Picking colors can be done in more than one way:
  - pick from the list provided, eg. `Color::dark_magenta` – note that all of the enum values are exported to the scope of the Color class, and are public
  - pick a colour from the palette of colours that are easy to display – numbered from 0-255 using `Color(100)` for example would give whichever color is the 100's one in our palette – we will do an example to draw the palette in a bit
  - pick a color using a number from the RGB color system (google "RGB colors")
- constructors for `Color` allow use of plain `int` or one of enum values from `Color_type`
- method `as_int()` is `const` since it doesn't change the color, and returns `int` value of color
-

Line_style

- We can also set the line style of `Shape` objects, such as making our grid have dashed lines: `grid.set_style( Line_style::dash );`

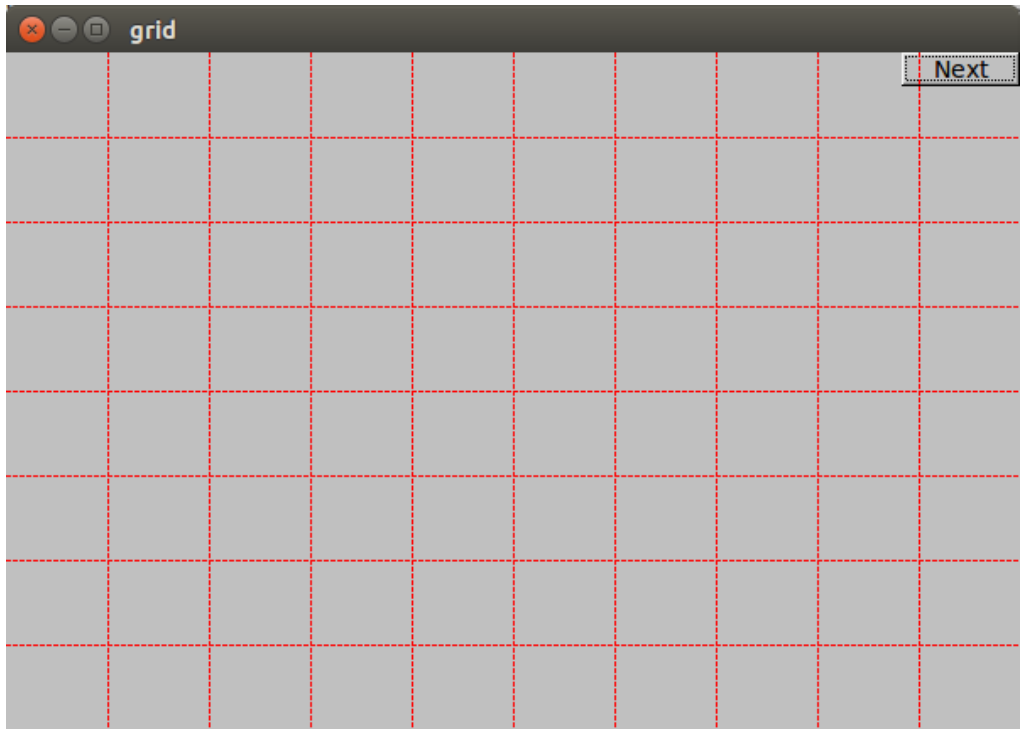

Figure : Making the grid dashed

Line_style interface

```
1  struct Line_style {
2    enum Line_style_type {
3       solid=FL_SOLID,                 // ————————
4       dash=FL_DASH,                   // — — — —
5       dot=FL_DOT,                     //  .......
6       dashdot=FL_DASHDOT,             // — . — .
7       dashdotdot=FL_DASHDOTDOT,       //  —..—..
8    };
9    Line_style( Line_style_type ss )
10       : s( ss ), w( 0 ) { }
11   Line_style( Line_style_type lst , int ww)
12       : s( lst ), w( ww ) { }
13   Line_style( int ss )
14       : s( ss ), w( 0 ) { }
15   int width() const { return w; }
16   int style() const { return s; }
17 private:
18   int s;
19   int w;
20 };
```

Notes on `Line_style`

- `Line_style` has two components
  1. A line width (default width is 1 pixel)
  2. A line style (default is Line_style::solid)
- Why hide the `int` used for line styles in `fltk`?
  - Next version of `fltk` could change to using `Fl_linestyle`
  - May want to adapt interface to use with different GUI library
  - Also makes clear that `int` has a particular meaning
- We can set both aspects of line style using `set_style`, for example, make the grid dotted and line thickness 2:

```
1 grid.set_style( Line_style{ Line_style::dot, 2} );
```

`Line_style` – dotted thicker grid

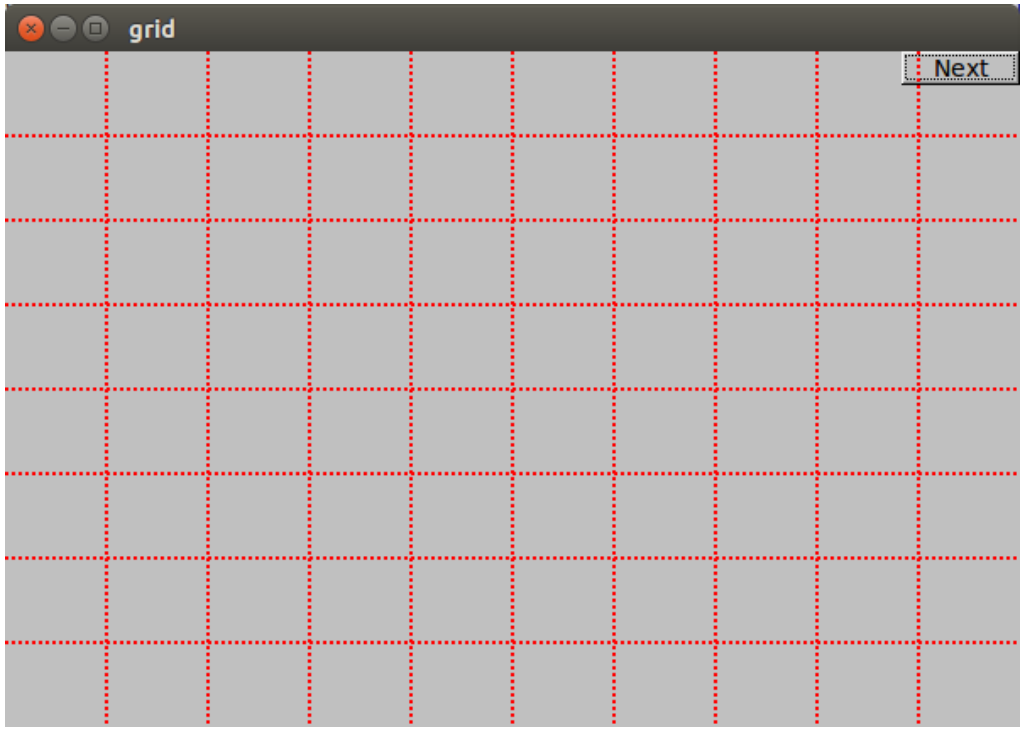

Figure : The dotted thicker grid

Open_polyline example use in drawPolylines.cpp

- Open_polyline objects are series of connected line segments
- Example, draw a square inward spiral:

```
1     // inward spiral, doing int N loops
2     Open_polyline opl;
3     const int offs=50; // offset into window
4     // distance between spirals
5     // (ww is width of screen, hh is height of
          screen):
6     int dsp = ( std::min<int>(ww,hh)-offs*2)/N/2;
7     for ( int i=0; i< N; ++i ){
8       // each loop has four lines (four points)
9       opl.add(Point{offs+dsp*i, offs+dsp*i});
10      opl.add(Point{offs+dsp*i, hh-offs-dsp*i});
11      opl.add(Point{ww-offs-dsp*i, hh-offs-dsp*i});
12      opl.add(Point{ww-offs-dsp*i, offs+dsp*i});
13      opl.add(Point{offs+dsp*(i+1), offs+dsp*i});
14    }
15    opl.set_style( Color::cyan );
16    win.attach( opl );
```

Result of `drawPolylines.cpp` with N=11

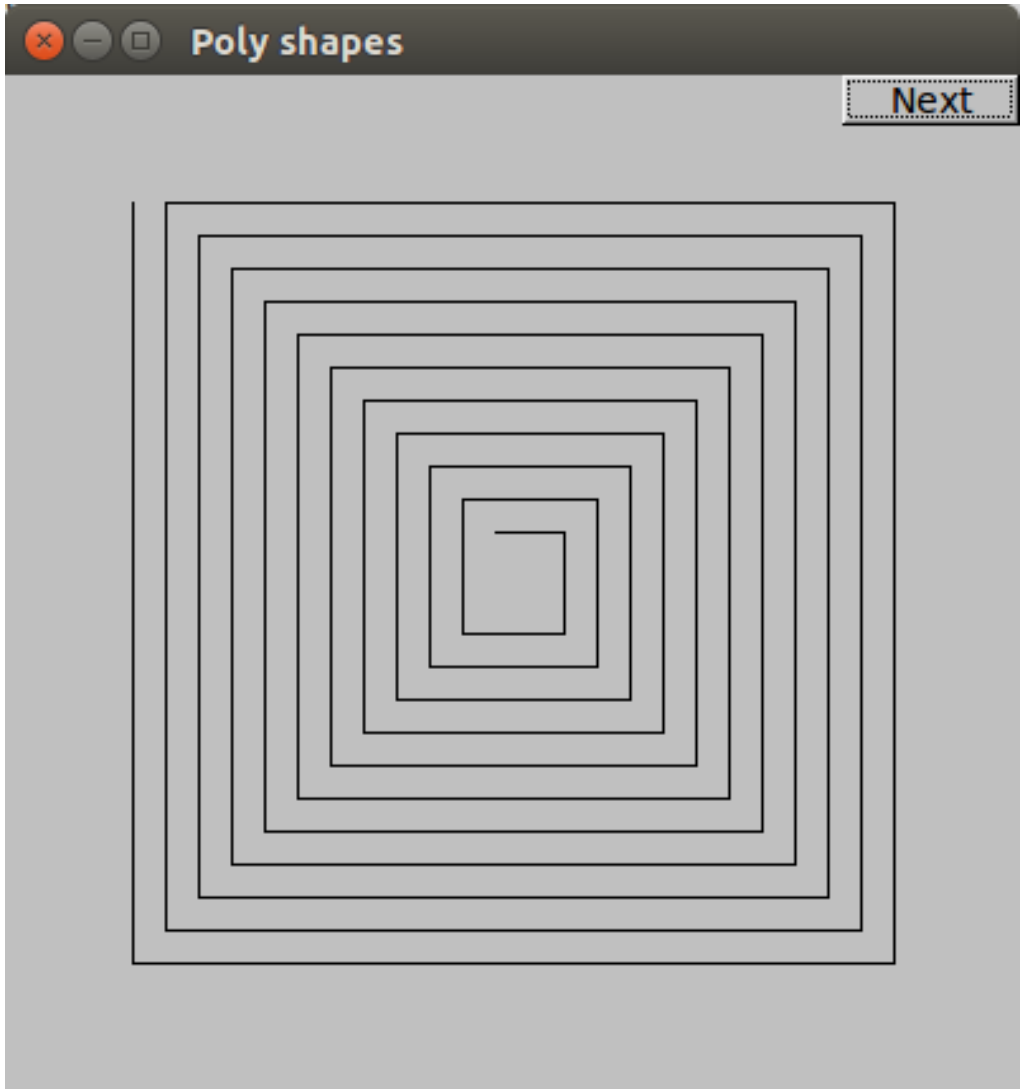

Figure : Inward spiral `Open_polyline`

Open_polyline interface

```
// Open sequence of lines
struct Open_polyline : Shape {
  using Shape::Shape; // use Shape's constructors
  void add(Point p) { Shape::add(p); }
  void draw_lines() const;
};
```

- `Open_polyline` inherits from `Shape`, and represents a sequence of points connected by lines
- The `using` declaration is shorthand for defining the constructors of `Open_polyline` to be the same as those in `Shape`
- `Shape` has a default constructor, and initializer-list constructor.
- the `add()` method of `Open_polyline` simply accesses the `Shape` class's `add()`

`Closed_polyline` example

- `Closed_polyline` has its last point connected to its first by a line, otherwise it is similar to `Open_polyline`
- Example: draw an N sided star (for N>=5).

```
Closed_polyline cpl;
// Corners of star will be at some radius r from
// center of star and equidistant angles 2pi/N.
// Convert polar coords r,theta to cartesian:
// x = r * cos( theta );    y = r * sin( theta );
int rstar = (std::min<int>(ww, hh) - offs*2)/2;
double thstar = 2.*acos(-1)/N; //angle pt is at
// always skip one point when going around star
for ( int i = 0; i < N; ++i ) {
  double curtheta = 2*i*thstar;
  int xx = rstar * cos( curtheta ) + rstar + offs;
  int yy = rstar * sin( curtheta ) + rstar + offs;
  cpl.add( Point{ xx, yy } );
}
cpl.set_style( Line_style{ Line_style::dot, 2 } );
cpl.set_color( Color::blue );
win.attach( cpl );
```

Result of `drawPolylines.cpp` with N=7

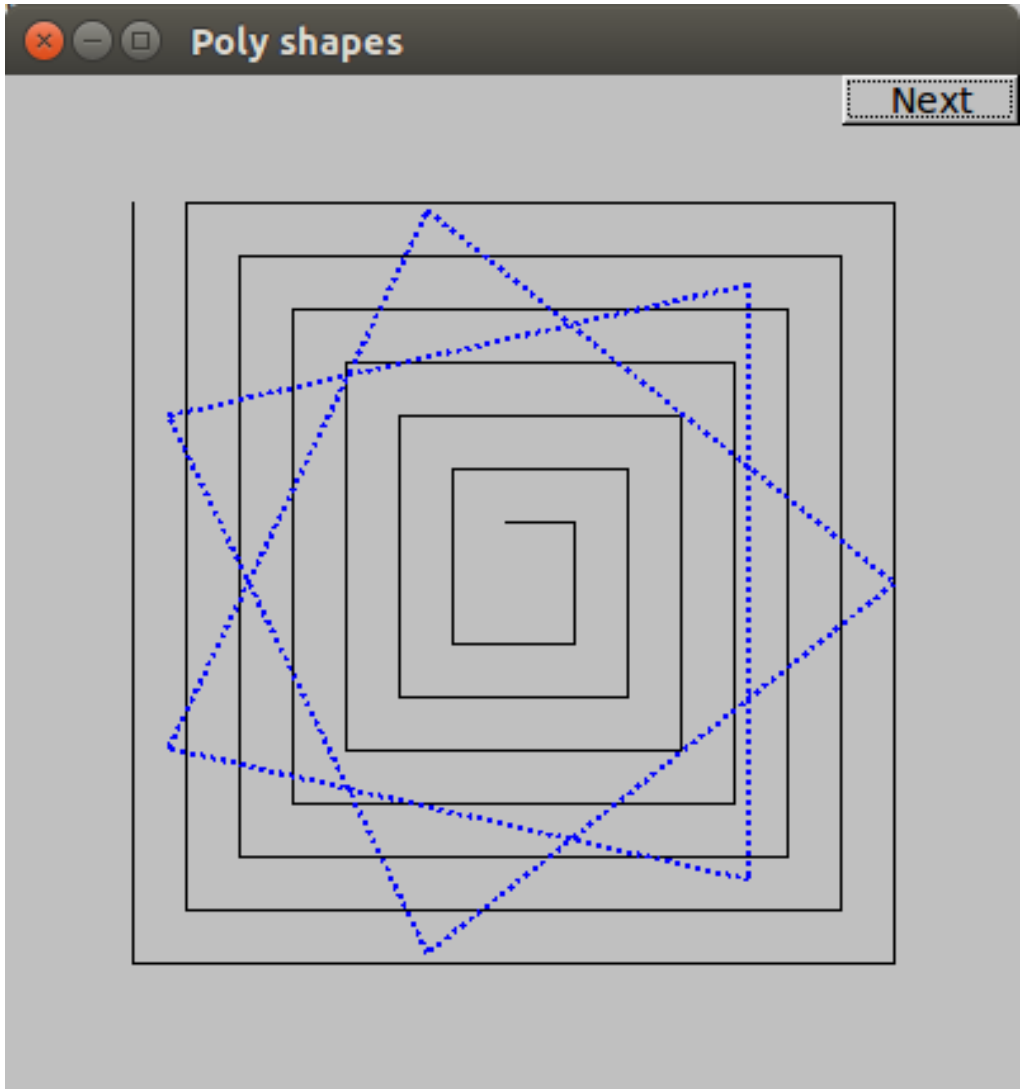

Figure : After attaching `Open_closedline` star

`Closed_polyline` interface and implementation

```
// Closed sequence of lines
struct Closed_polyline : Open_polyline {
  // Uses same constructors as Open_polyline
  using Open_polyline::Open_polyline;
  // has its own way of drawing lines, ie
  // also draws line between first and last point
  void draw_lines() const;
};
// Note implementation of draw_lines() first calls
// one for Open_polyline:
void Closed_polyline :: draw_lines(){
  Open_polyline::draw_lines();
  // then draw closing line
  if ( color().visibility() ){
    fl_line( point(number_of_points()-1).x,
             point(number_of_points()-1).y,
             point(0).x,
             point(0).y);
  }
}
```

Notes on `Closed_polyline`

- By using inheritance, we only need to program what is different between `Closed_polyline` and `Open_polyline`
- Implementation uses `FLTK`'s line drawing function
- Note that this detail is kept out of the interface, so the user need not see how the line drawing is implemented
- We could therefore replace `FLTK` in the implementation, and the user need not know that anything has changed

Polygon

- `Polygon` is very similar to `Closed_polyline`
- Only difference is `Polygon` is not allowed to have lines that cross, which is not allowed in classic definition of polygons
- Interface looks like:

```
1 // closed sequence of non-intersecting lines
2 struct Polygon : Closed_polyline {
3   using Closed_polyline::Closed_polyline;
4   void add(Point p);
5   void draw_lines() const;
6 };
```

- Note that it inherits constructors from `Closed_polyline`
- `draw_lines()` method just calls `Closed_polyline::draw_lines()`
- Main difference is `add()` method needs to check if newly added line intersects any of existing lines.
- For a polygon of N sides, needs to call function to check intersection $N * (N - 1)/2$ times.
- Best to use `Polygon` only for few sides (eg. for $N = 100$, would check intersections 4550 times)

Example use of `Polygon` in `drawPolylines.cpp`

- Lets draw an N-sided polygon around our N-sided star.

```
// (ww is width of screen, hh is height of screen):
int rstar = (std::min<int>( ww, hh ) - offs*2)/2;
double thstar = 2.*acos(-1)/N; //ang b/w points
Polygon poly;
// points at same place as star,
// just don't skip points going around shape
for ( int i = 0; i < N; ++i ) {
  double curtheta = i*thstar;
  int xx = rstar * cos( curtheta ) + rstar + offs;
  int yy = rstar * sin( curtheta ) + rstar + offs;
  poly.add( Point{ xx, yy } );
}
poly.set_style( Line_style{Line_style::dash, 2} );
poly.set_color( Color::red );
win.attach( poly );
```

Result of `drawPolylines.cpp` with N=9

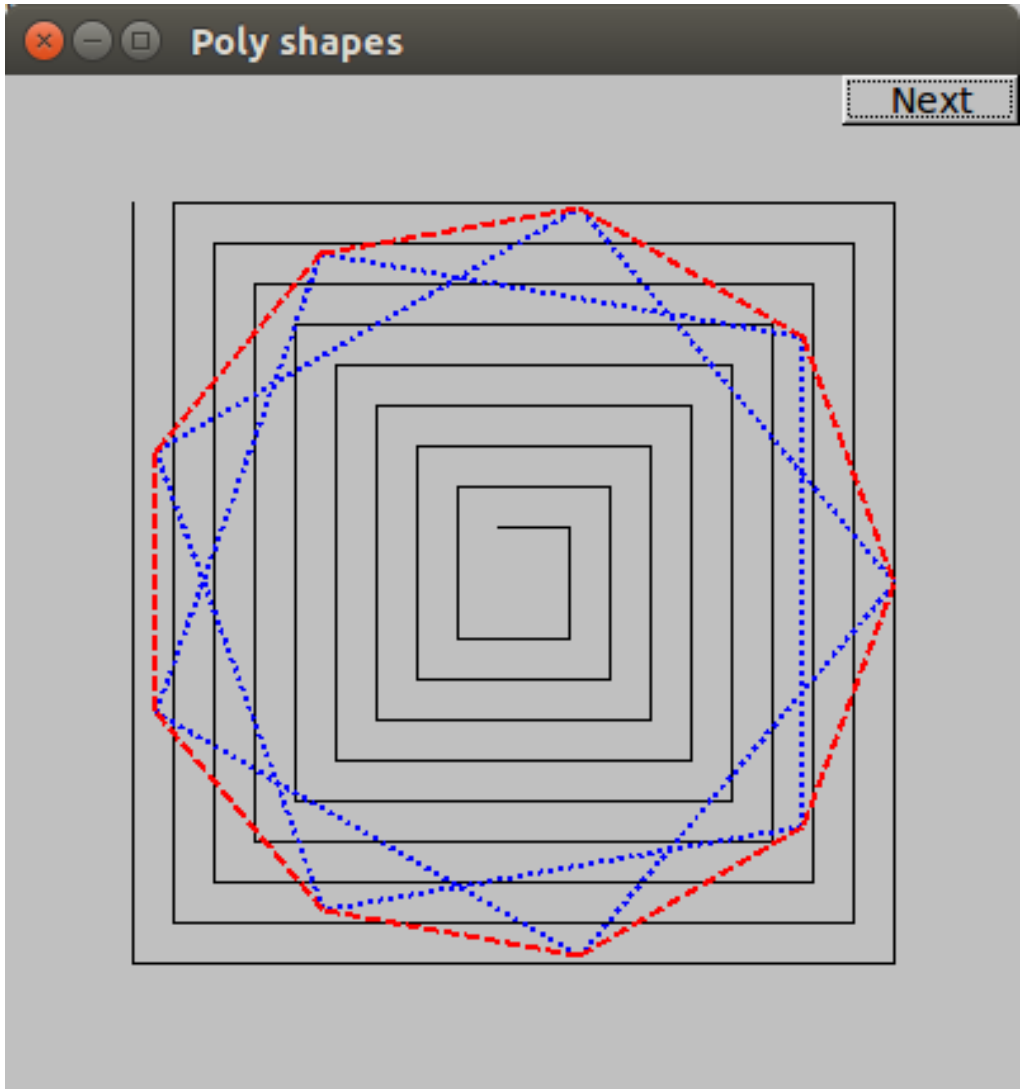

Figure : After attaching `Polygon` around star

`Rectangle` Interface

- Rectangles are so commonly used, they are given their own shape:

```
struct Rectangle : Shape {
  Rectangle(Point xy, int ww, int hh);
  Rectangle(Point x, Point y);
  void draw_lines() const;
  int height() const { return h; }
  int width() const { return w; }
private:
  int w; // width
  int h; // height
};
```

- Rectangle is specified in one of two ways:
  1. By top left and bottom right `Points`
  2. By top left point, width as int, and height as int

`Rectangle` constructors

```
1  // given top left point, width and height
2  // just assign values, only if top left point is okay
3  Rectangle::Rectangle(Point xy, int ww, int hh)
4    : w{ ww }, h{ hh }  {
5    if (h<=0 || w<=0)
6      error("Bad rectangle: non-positive side");
7    add(xy);
8  }
9  // given two points, need to calculate width and height
10 // again check top left point is okay
11 Rectangle::Rectangle(Point x, Point y)
12   : w{ y.x - x.x }, h{ y.y - x.y }  {
13   if (h<=0 || w<=0)
14     error("Bad rectangle: first point not top left");
15   add(x);
16 }
```

- Note that top left corner gets added to list of points inherited from `Shape` class

`Rectangle` example in `drawRectangles.cpp`

- Another reason for having rectangle as a separate shape is that it is easiest shape to fill with colour
- For example we can build several rectangles with different fill colors:

```
1 Rectangle r00{ Point{150,100}, 200, 100 };
2 Rectangle r11{ Point{ 50, 50}, Point{250,150} };
3 Rectangle r12{ Point{ 50,150}, Point{250,250} };
4 Rectangle r21{ Point{250, 50}, 200, 100 };
5 Rectangle r22{ Point{250,150}, 200, 100 };
6 r00.set_fill_color( Color::red );
7 r11.set_fill_color( Color::yellow );
8 r12.set_fill_color( Color::green );
9 r21.set_fill_color( Color::blue );
```

Result of `drawRectangles.exe`

- Note that when no fill color has been set, the rectangle is transparent.

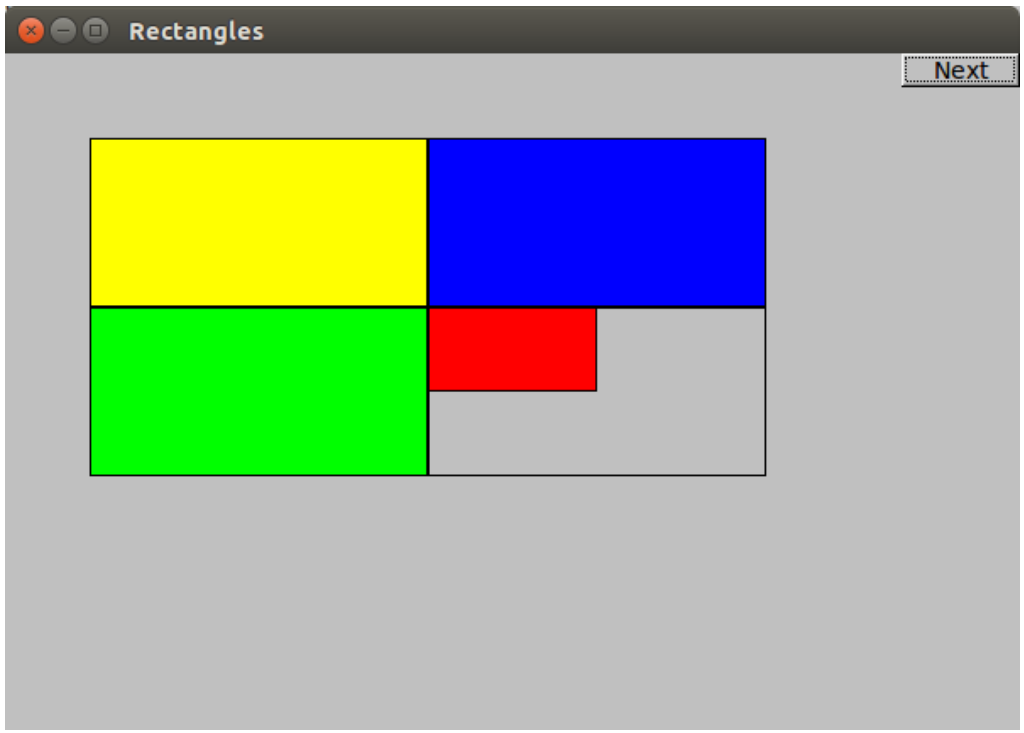

Figure : Result of running `drawRectangles.exe`

Moving objects on the window

- We can move and change the color of our rectangles:

```
1 // move to right by 400 pixels:
2 r11.move(400,0);
3 r11.set_fill_color(Color::white);
4 win.set_label("Moved the rectangle over");
```

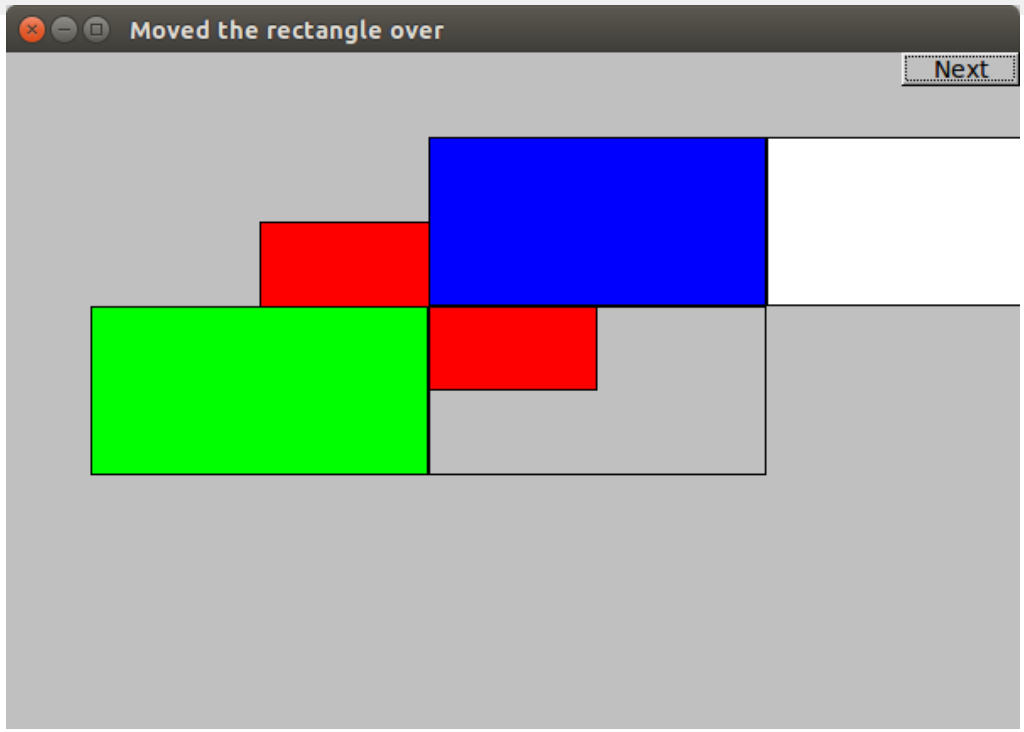

Figure : Note how white box gets clipped

Changing order of drawing objects

- Note that objects attached to window are drawn in the order they were attached
- Lines of rectangles are black and visible by default, can either change their color to match fill color, or set them to be invisible
- We can tell window to put a shape on top by calling:

```
1 win.put_on_top( r00 );
2 win.set_label("Yellow rectangle on top");
3 r00.set_color( Color::invisible );
4 r11.set_color( Color::invisible );
5 r12.set_color( Color::invisible );
6 r21.set_color( Color::invisible );
7 r22.set_color( Color::invisible );
```

Final result of `drawRectangles.cpp`

- Note that if we set both the line color and fill color to invisible, then the rectangle will not be seen

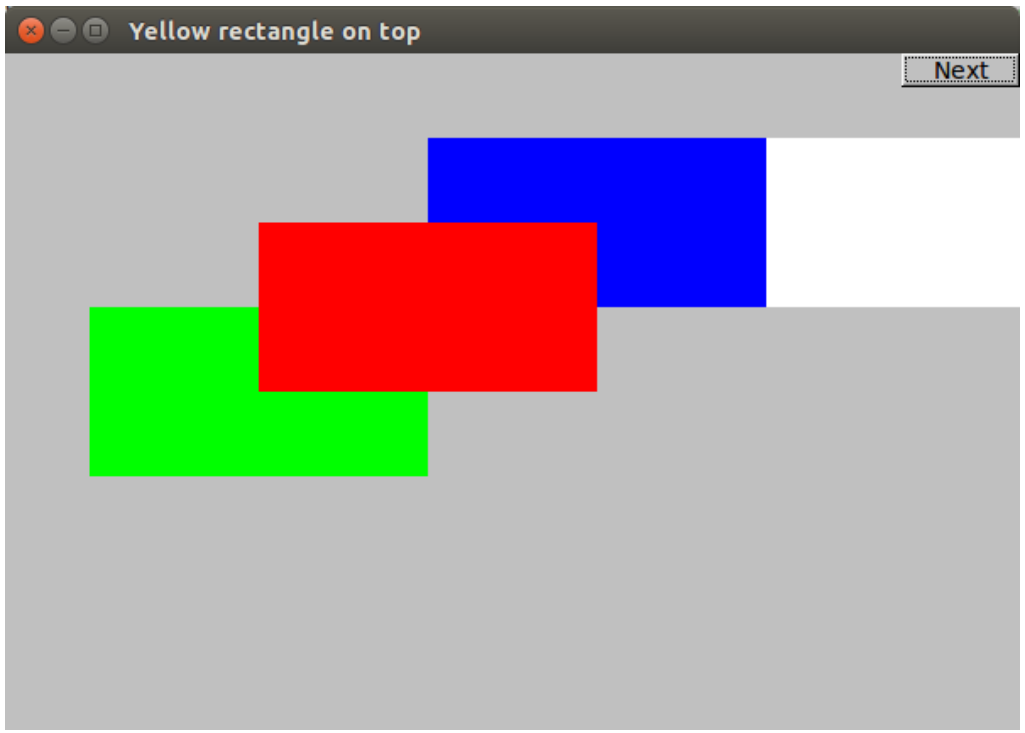

Figure : Result of `drawRectangles.exe`

Implementation of `Rectangle::draw_lines()`

```
1  void Rectangle::draw_lines() const
2  {
3    if ( fill_color().visibility() ) {
4      fl_color( fill_color().as_int() );
5      fl_rectf( point(0).x, point(0).y, w, h);
6      // return color to default
7      fl_color( color().as_int() );
8    }
9    if ( color().visibility() ) {
10     fl_color( color().as_int() );
11     fl_rect( point(0).x, point(0).y, w, h);
12   }
13 }
```

- Note that `fltk` provides function to fill rectangle `fl_rectf()` and one to draw its outline `fl_rect()`

Using objects without naming them

- So far we have made C++ names for each of the graphics objects that we drew
- If there are lots of objects to draw this can become impractical
- Instead we can keep a collection of graphics objects in a `vector`
- Note if all our objects are the same we can use standard library, eg for Rectangle objects: `std::vector<Rectangle>`
- We may want to have a container to hold multiple different types of graphics objects

Collection of elements, all the same

- Our `Vector_ref` container class is similar to a `std::vector`, and has interface:

```
template<class T> class Vector_ref {
  public:
    Vector_ref() {}
    Vector_ref(T* a, T* b=0, T* c=0, T* d=0);
    ~Vector_ref();
    void push_back(T&);
    void push_back(T*);
    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const;
  //...
  };
```

Example use of `Vector_ref`

- Used in a similar way to `std::vector`:

```
1  Vector_ref< Rectangle > rect;
2  // make a rectangle named r
3  Rectangle r{ Point{100, 200}, Point{200, 300} };
4  // add named rectangle to rect
5  rect.push_back( r );
6  // add un-named rectangle
7  // using C++ new'' operator to create new object
8  rect.push_back( new Rectangle{ Point{50,60},
       Point{80,90} } );
9  // Do things to objects in rect
10 for ( int i=0; i<rect.size(); ++i ){
11   rect[i].move( 10, 10 );
12 }
```

Notes on use of `Vector_ref`

- To make an un-named object we used the `new` operator, which allocates memory for an object that we need to eventually `delete`
- When we add this `new` object to `Vector_ref` we transfer ownership of that memory to the `Vector_ref` object – and it can deal with the deletion of the memory when the object is no longer being used
- The `new` operator returns a pointer to the memory that was allocated for the object of the type after `new`
- In this case we get back a pointer to a `Rectangle` type object
- We will discuss the `new` and `delete` operators in more detail when we get to memory management
- For now these details of the implementation of `Vector_ref` can remain hidden

Using un-named objects to create a palette

- Lets produce a palette of the colors using a 16×16 grid

`vectRefPalette.cpp`

```
Vector_ref<Rectangle> rect;
// offset into window
const int offs=50;
// for loop on i labels row
for ( int i = 0 ; i < 16 ; ++i ){
  // for loop on j labels column
  for ( int j = 0 ; j < 16 ; ++ j ){
    // add rectangle at new location
    rect.push_back( new Rectangle{
      Point{ offs+i*20, offs+j*20}, 20, 20 } );
    // set rectangle fill color
    rect[ rect.size()-1 ] . set_fill_color( Color{
        i*16+j } );
    // attach rectangle to window
    win.attach( rect[ rect.size()-1 ] );
  }
}
```

Result of running `vectRefPalette.cpp`

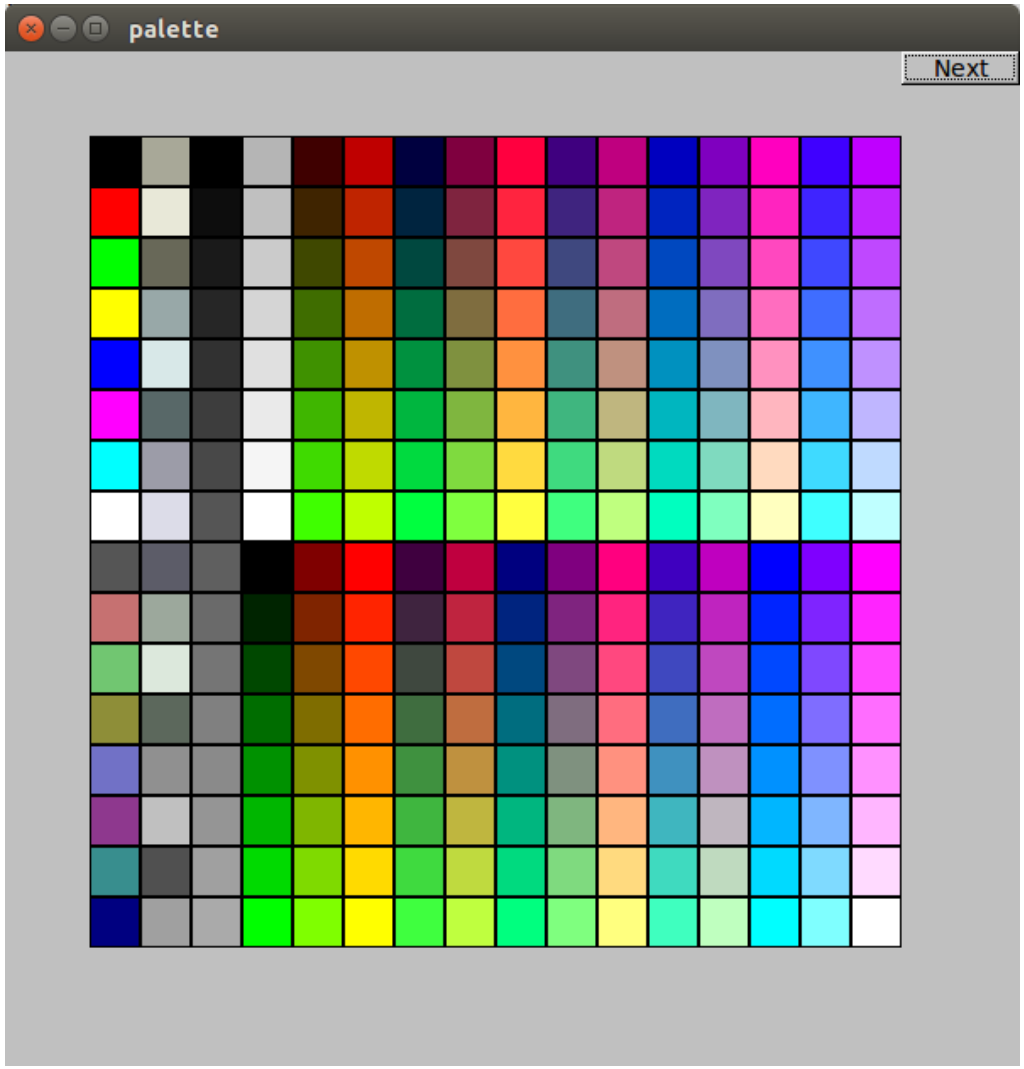

Figure : Result of running `vectRefPalette.exe`

Graphics class `Text`

- Text is often a part of displays
- For example we want to label the color numbers
- For example add a label next to a particular color:

```
1 // Example 0xcf is a lovely pink
2 Text t{ Point{100,570}, "<- Lovely pink is 0xa8" };
3 t.set_color( Color( 0xa8 ) );
4 win.attach( t );
5 Rectangle r{ Point{40,550}, recsize, recsize } );
6 r.set_fill_color( Color( 0xa8 ) );
7 win.attach( r );
```

- `Point` defines lower left corner of where the text will be placed on the window

Making palette useful for looking up colors

- We can finish labelling the palette with the numbers of the colors in hexadecimal notation:

```
1  // build up labels to put hex numbers to colors
2  Vector_ref< Text > text;
3  for ( int i = 0 ; i < 16 ; ++i ){ //row
4    ostringstream oss;
5    oss << hex << setw(1) << i;
6    text.push_back( new Text{
7         Point{ offs+10+i*recsize , offs -10},
               oss.str() } );
8    win.attach( text[ text.size()-1 ] );
9  }
10 for ( int i = 0 ; i < 16 ; ++i ){ //column
11   ostringstream oss;
12   oss << hex << setw(2) << setfill('0') << i*16;
13   text.push_back( new Text{
14        Point{ offs/2, offs+offs/2+i*recsize},
               oss.str() } );
15   win.attach( text[ text.size()-1 ] );
16 }
```

Final result of our color lookup table

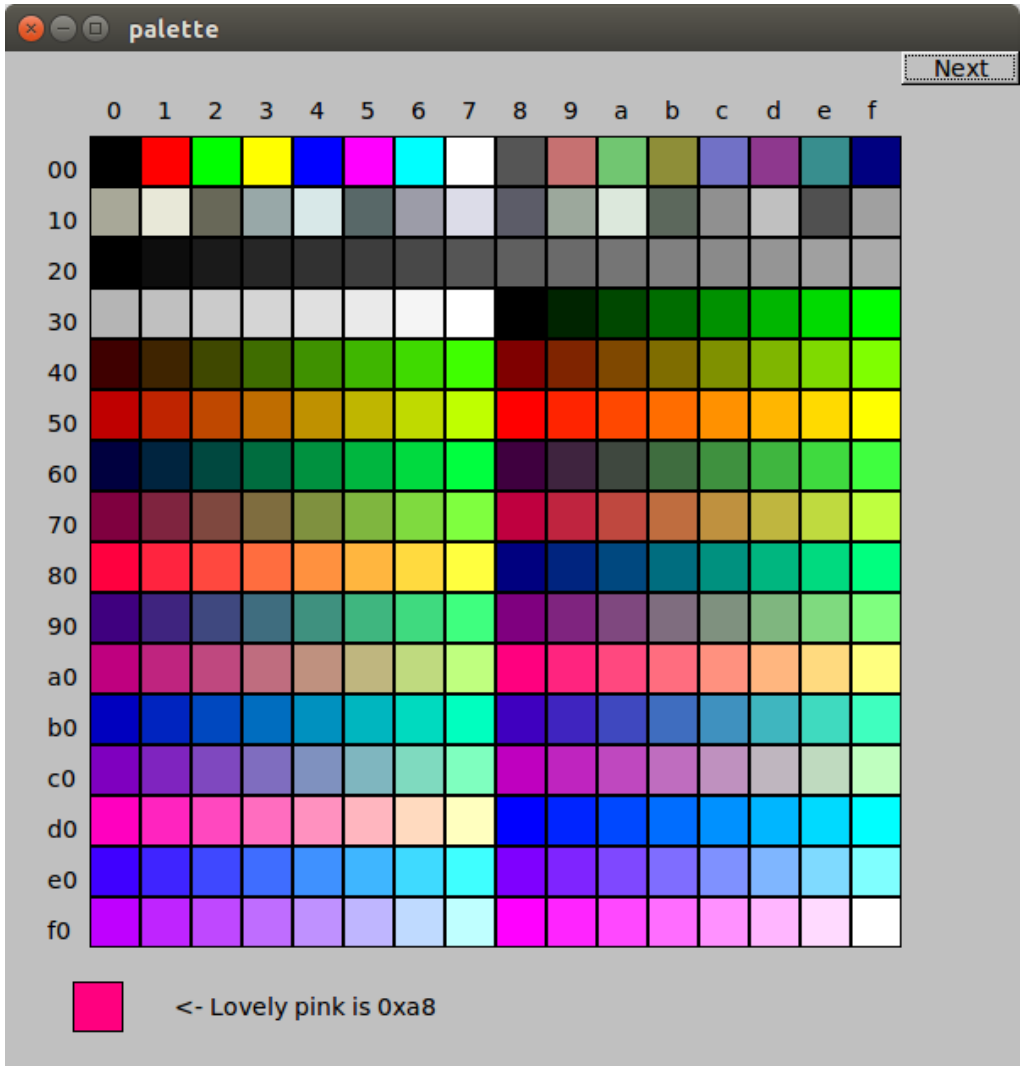

Figure : Final result of running `vectRefPalette.exe`

Text interface

```
struct Text : Shape {
    Text{ Point x, const string & s ) :
      lab( s ) { add( x ); }
    void draw_lines() const;
    void set_label( const string & s ){ lab = s; }
    string label() const { return lab; }
    void set_font(Font f) { fnt = f; }
    Font font() const { return Font(fnt); }
    void set_font_size(int s) { fnt_sz = s; }
    int font_size() const { return fnt_sz; }
  private:
    string lab; // label
    Font fnt{ fl_font() };
    // at least 14 point
    int fnt_sz{ (14<fl_size()) ? fl_size() : 14 };
  };
}
```

Notes on `Text` interface

- Note default value of `fnt_sz` is set to 14 in case the `fltk` default is smaller
- `Text` has its own `draw_lines`, since text is obviously drawn differently than lines, and is implemented as:

```
1 void Text::draw_lines() const {
2   fl_draw( lab.c_str(), point(0).x, point(0).y );
3 }
```

- Color of the line is determined same way as it is for shapes, and can be set with the `Shape::set_color` method that it inherits
- Text has a particular font and font-size. The type of font, similar to the way Color is done, is a class to map the font types in `fltk` to our interface.

`Font` interface – very analogous to `Color`

```
class Font { public:
 enum Font_type {
   helvetica=FL_HELVETICA,
   helvetica_bold=FL_HELVETICA_BOLD,
   helvetica_italic=FL_HELVETICA_ITALIC,
   helvetica_bold_italic=FL_HELVETICA_BOLD_ITALIC,
   courier=FL_COURIER, courier_bold=FL_COURIER_BOLD,
   courier_italic=FL_COURIER_ITALIC,
   courier_bold_italic=FL_COURIER_BOLD_ITALIC,
   times=FL_TIMES, times_bold=FL_TIMES_BOLD,
   times_italic=FL_TIMES_ITALIC,
   times_bold_italic=FL_TIMES_BOLD_ITALIC,
   symbol=FL_SYMBOL, screen=FL_SCREEN,
   screen_bold=FL_SCREEN_BOLD,
   zapf_dingbats=FL_ZAPF_DINGBATS
 };
 Font(Font_type ff) :f(ff) { }
 Font(int ff) :f(ff) { }
 int as_int() const { return f; }
private:
 int f;
};
```

Circle

- A `Circle` class has interface:

```
struct Circle : Shape {
  Circle(Point p, int rr);  // center and radius
  void draw_lines() const;
  Point center() const;
  void set_radius(int rr);
  int radius() const;
private:
  int r;
};
```

Example use of `Circle`

- Draw three different sized circles
- We specify the center coordinates of the circle, and its radius

circleTests.cpp

```
1 Circle c1{ Point{100, 200}, 50 };
2 Circle c2{ Point{150, 200},100 };
3 Circle c3{ Point{200, 200},150 };
4 win.attach(c1);
5 win.attach(c2);
6 win.attach(c3);
```

Circles drawn by `circleTests.cpp`

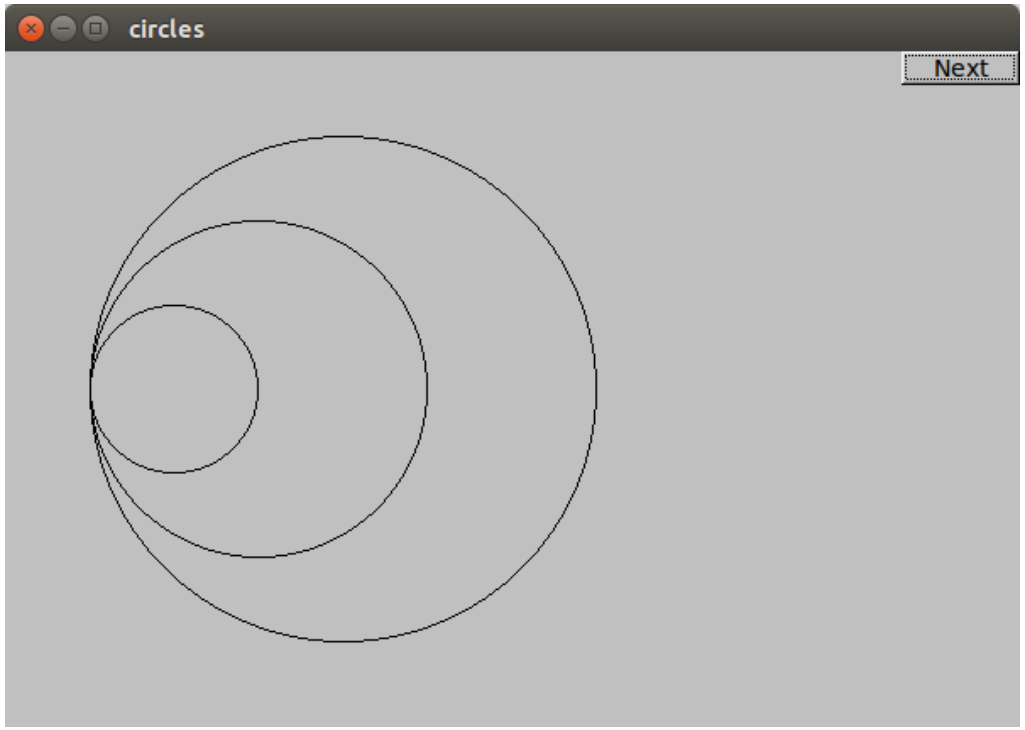

Figure : Result of running `circleTests.exe`

`Circle` implementation

- Main peculiarity is that top left corner of circle is stored rather than center

```
1  Circle::Circle(Point p, int rr){
2        :r{ rr } {
3     add(Point{ p.x - r, p.y - r });
4  }
5  Point center() const {
6    return { point(0).x + r, point(0).y + r };
7  }
8  void set_radius(int rr) {
9    // maintain center of circle
10   set_point(0, center().x-rr, center().y-rr);
11   r=rr;
12 }
```

`Circle::draw_lines` implementation

- To draw the circle, the interface class uses `fltk`'s function to draw arcs (`fl_arc()`) for the outline, and `fl_pie` for filling the circle
- We draw whole circle, but `fltk` allows drawing just an arc of a circle

```
1  void Circle::draw_lines() const
2  {
3      if (fill_color().visibility()) { // fill
4          fl_color(fill_color().as_int());
5          fl_pie(point(0).x,point(0).y,r+r-1,r+r-1,0,360);
6          fl_color(color().as_int()); // reset color
7      }
8
9      if (color().visibility()) { // outline
10         fl_color(color().as_int());
11         fl_arc(point(0).x,point(0).y,r+r,r+r,0,360);
12     }
13 }
```

Ellipse

- An ellipse has a major and minor axis instead of radius
- To define an ellipse we provide:
  - center coordinate
  - distance from center point to ellipse along its x-axis, and
  - distance from center point to ellipse along its y-axis

```
1  struct Ellipse : Shape {
2    // center, radius in x, and radius in y
3    Ellipse(Point p, int ww, int hh);
4    void draw_lines() const;
5    Point center() const;
6    Point focus1() const;
7    Point focus2() const;
8    void set_major(int ww);
9    int major() const;
10   void set_minor(int hh);
11   int minor() const;
12 private:
13   int w;  int h;
14 };
```

Example use of `Ellipse`

- Three ellipses with common center but different length axes:

```
1 Ellipse e1{ Point{450,200}, 50, 50};
2 Ellipse e2{ Point{450,200},100, 50};
3 Ellipse e3{ Point{450,200},100,150};
4 win.attach( e1 );
5 win.attach( e2 );
6 win.attach( e3 );
```

Ellipses drawn by updated `circleTests.cpp`

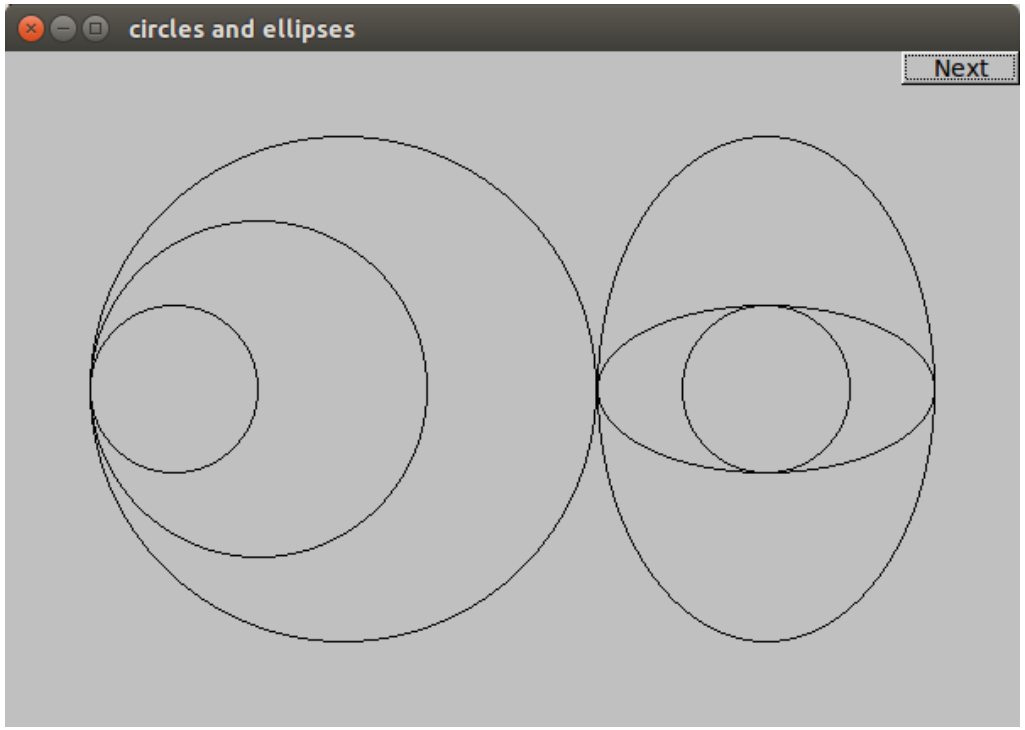

Figure : Result of running updated `circleTests.exe`

Notes on `Ellipse`

- Note that an ellipse with same lengths for major and minor axes looks exactly like a circle
- Ellipses are often defined in terms of two focus points plus a sum of distances from a point to the focus.
- Methods are provided to get the focus of the ellipse, for example:

```
1 Point Ellipse::focus1() const {
2   // return the focus along the x-axis
3   return{ center().x +
4     int( sqrt(double(w*w - h*h))), center().y };
5 }
```

Why have separate `Circle` and `Ellipse`

- We could define circles using `Ellipse` – every circle is an ellipse, but not every ellipse is a circle
- Maybe add a method to check if the `Ellipse` is a circle by checking if the two radii are the same
- Primary reason to have Circle is that ellipse needs extra memory to store the location of an extra radius (along a second axis)
- Also if we defined a circular shape using `Ellipse` could always call `set_minor` or `set_major` to change the circle into an ellipse
- A `Circle` could never morph into an ellipse.

Designing classes

- Be careful to design classes that represent a coherent concept rather than just a collection of data and classes
- Just throwing code together without thinking ("hacking") can lead to code that is hard to maintain
- Such code is often difficult to debug and for others to maintain and use
- *Remember that others might be you in a few months*

Marked_polyline

- Sometimes we may want the points at the ends of line segments to be drawn with a marker – for that there is the `Marked_polyline`:

```
1 Marked_polyline bigdipr{"O*OxooO"};
2 bigdipr.add( Point{ 70, 200 } );
3 bigdipr.add( Point{140, 160 } );
4 bigdipr.add( Point{200, 180 } );
5 bigdipr.add( Point{270, 200 } );
6 bigdipr.add( Point{300, 250 } );
7 bigdipr.add( Point{400, 220 } );
8 bigdipr.add( Point{395, 150 } );
9 win.attach( bigdipr );
```

- Note use of string to represent marker for each point
- If not provided defaults to '*' for all points

Result of `markerTests.cpp`

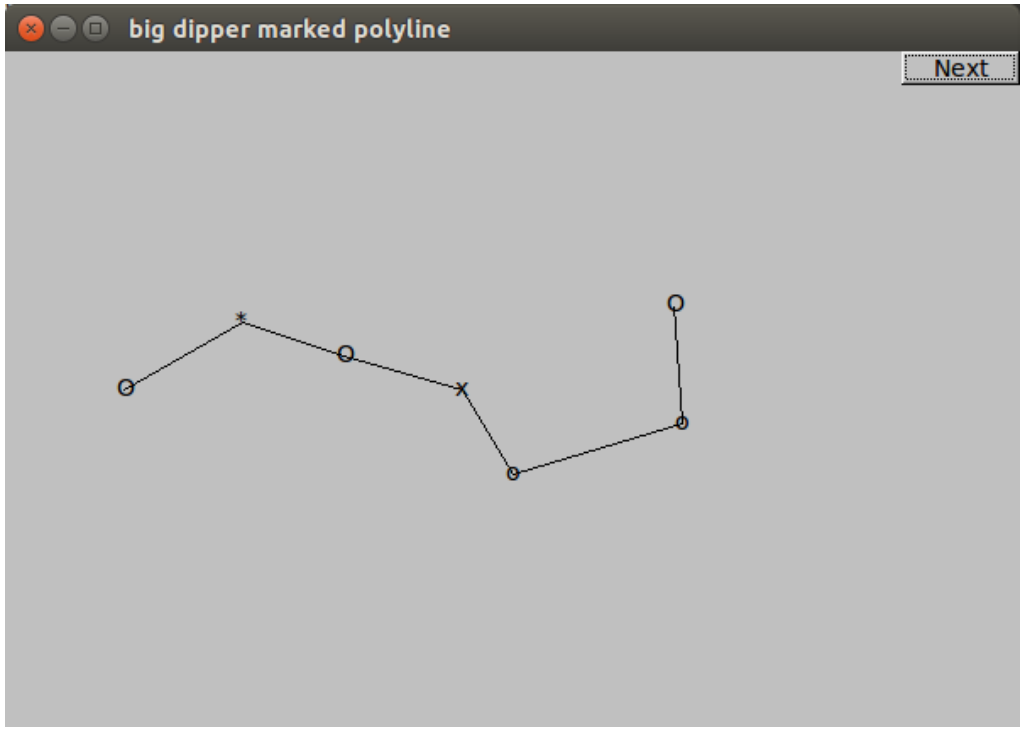

Figure : Result of running `markerTests.exe`

Interface to `Marked_polyline`

```
1 struct Marked_polyline : Open_polyline {
2   Marked_polyline(const string& m) :mark(m) { }
3   void draw_lines() const;
4 private:
5   string mark;
6 };
```

- By inheriting from `Open_polyline`, lines are already handled
- Only have to add the marks

Drawing the `Marked_polyline`

```
1 void Marked_polyline::draw_lines() const {
2   Open_polyline::draw_lines();
3   for (int i=0; i<number_of_points(); ++i) {
4     draw_mark(point(i),mark[i%mark.size()]);
5   }
6 }
```

- First calls `Open_polyline::draw_lines()`, to handle drawing the lines between points
- Then loops over points to draw mark at each point
- `mark[i%mark.size()]` selects character from string of marks – and modulo operator makes cycle of marks repeat

Drawing the `Marked_polyline` continued

- `draw_lines` uses a helper function `draw_mark()` to output the letter at a given point:

```
1 void draw_mark(Point xy, char c) {
2   static const int dx = 4;
3   static const int dy = 4;
4   string m(1,c);
5   fl_draw( m.c_str(), xy.x-dx, xy.y+dy );
6 }
```

- `dx, dy` are used as constants to center letter over point
- string `m` is constructed to hold the single letter marker
- calls `fl_draw` function from `fltk` library

Marks

- For displaying marks without connecting lines between them
- For example the same “Big dipper” without connecting lines and offset by a bit:

```
Marks bigdip{"O*OxooO"};
bigdip.add( Point{ 70, 300 } );
bigdip.add( Point{140, 260 } );
bigdip.add( Point{200, 280 } );
bigdip.add( Point{270, 300 } );
bigdip.add( Point{300, 350 } );
bigdip.add( Point{400, 320 } );
bigdip.add( Point{395, 250 } );
win.attach( bigdip );
```

Updated result of `markerTests.cpp`

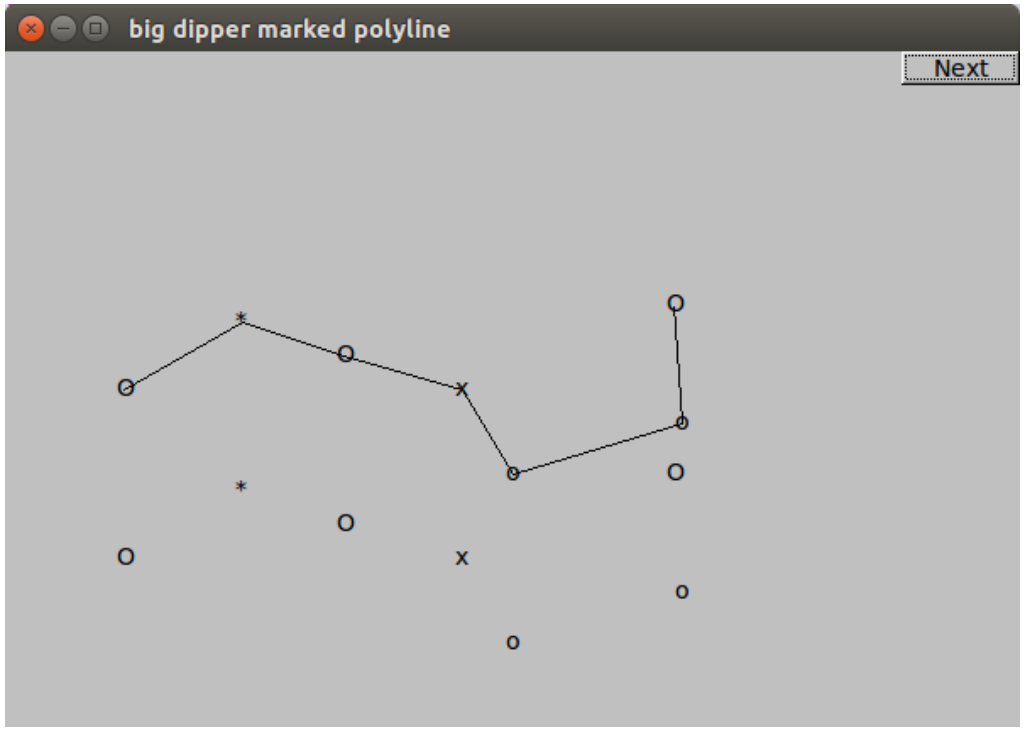

Figure : Result of running `markerTests.exe`

Marks

- `Marks` is just a `Marked_polyline` with its lines made `invisible`:

```
1 struct Marks : Marked_polyline {
2   Marks(const string& m) :Marked_polyline(m) {
3     set_color(Color(Color::invisible)); }
4 };
```

Mark

- `Mark` is a `Marks` with just a single mark

```
1 struct Mark : Marks {
2   Mark(Point xy, char c) : Marks(string(1,c))
        {add(xy); }
3 };
```

- The expression `string(1,c)` is a constructor for `string` that contains a single character `c`
- Was it worth the effort adding `Mark`?
- Answer is not clear.

Images

- Fairly minimal support for images is provided
- Allows selecting part of image to draw

```
1 Image canada{Point{0,0},"canada.jpg"};
2 // shift map up a bit and apply width and height
      window
3 canada.set_mask( Point{0,100}, 600,400 );
4 Image manitoba{Point{400,0},"manitoba.jpg"};
5 win.attach( canada );
6 win.attach( manitoba );
```

- `set_mask()` operation selects part of image to be displayed
- in example above 600x400 pixel region is selected in the canada.jpg image

Result of `imageTests.cpp`

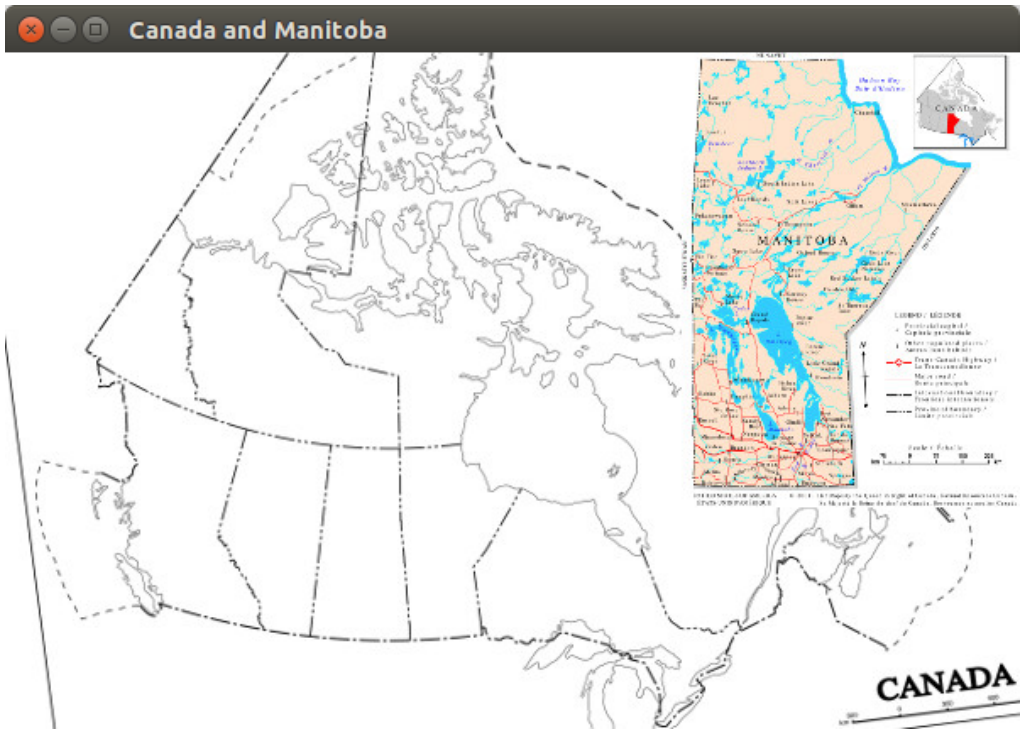

Figure : Result of running `imageTests.exe`

Image class interface

- Allow for jpg and gif image, and keep track of type with `enum class Suffix { none, jpg, gif};`
- Image in memory is represented by class `Image`:

```
1  struct Image : Shape {
2    Image(Point xy, string s, Suffix::Encoding e =
         Suffix::none);
3    ~Image() { delete p; }
4    void draw_lines() const;
5    void set_mask(Point xy, int ww, int hh)
6      { w=ww; h=hh; cx=xy.x; cy=xy.y; }
7  private:
8    // define "masking box" within image relative to
         position (cx,cy)
9    int w,h,cx,cy;
10   Fl_Image* p;
11   Text fn;
12 };
```

`Image` notes

- images are quite complex to handle
- constructor checks file with image can be opened, and decides the image type
- if file is *not* of correct format, it sets image to be bad
- The definition of `Bad_image` is:

```
1 struct Bad_image : Fl_Image {
2   Bad_image(int h, int w) : Fl_Image(h,w,0) { }
3   void draw(int x,int y, int, int, int, int) {
        draw_empty(x,y); }
4 };
```

Image constructor

```
1  // somewhat over elaborate constructor
2  // errors related to image files are pain to debug
3  Image::Image(Point xy, string s, Suffix::Encoding e)
4  :w(0), h(0), fn(xy,"") {
5    add(xy);
6    if (!can_open(s)) {
7      fn.set_label("cannot open \""+s+'\"');
8      p = new Bad_image(30,20); // the "error image"
9      return;
10   }
11   if (e == Suffix::none) e = get_encoding(s);
12   switch(e) {
13     case Suffix::jpg:
14       p = new Fl_JPEG_Image(s.c_str()); break;
15     case Suffix::gif:
16       p = new Fl_GIF_Image(s.c_str()); break;
17     default: // Unsupported image encoding
18       fn.set_label("unsupported file type \""+s+'\"');
19       p = new Bad_image(30,20); // the "error image"
20   }
21 }
```

Notes on `Image` constructor

- We use the file suffix to choose the type of image file
- Either calls the `fltk` function `Fl_JPEG_Image` or `Fl_GIF_Image`
- Again we see use of `new` operator, this time to allocate memory for the image
- Also use a number of helper functions:
  - `get_encoding()` which tries to determine the file time from the extension
  - `can_open()` which tries to open the file and returns false if it cannot be opened

Week 9 Done!

- Homework 6 given out today is due Nov. 11 (17:00)
- Homework 7 given out today is due Nov. 18 (17:00)
- Reminder: final project is due Nov. 29 (17:00)