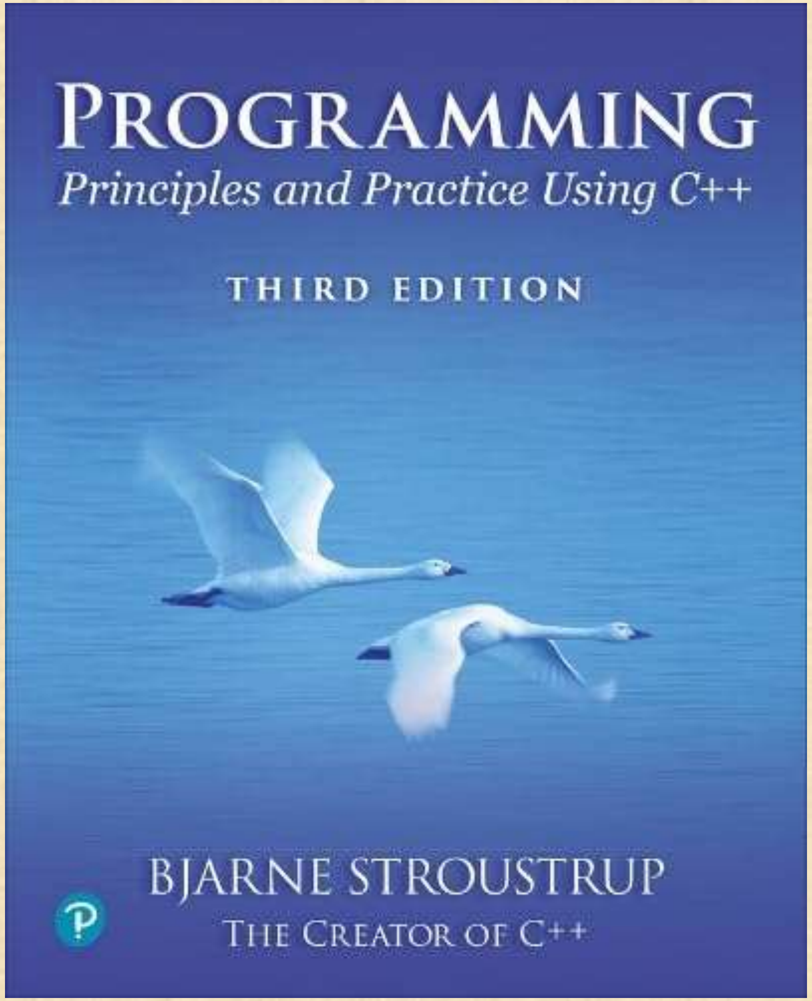
Chapter 11 – Graphics Classes



A language that doesn´t change the way you think isn´t worth learning.
– Traditional

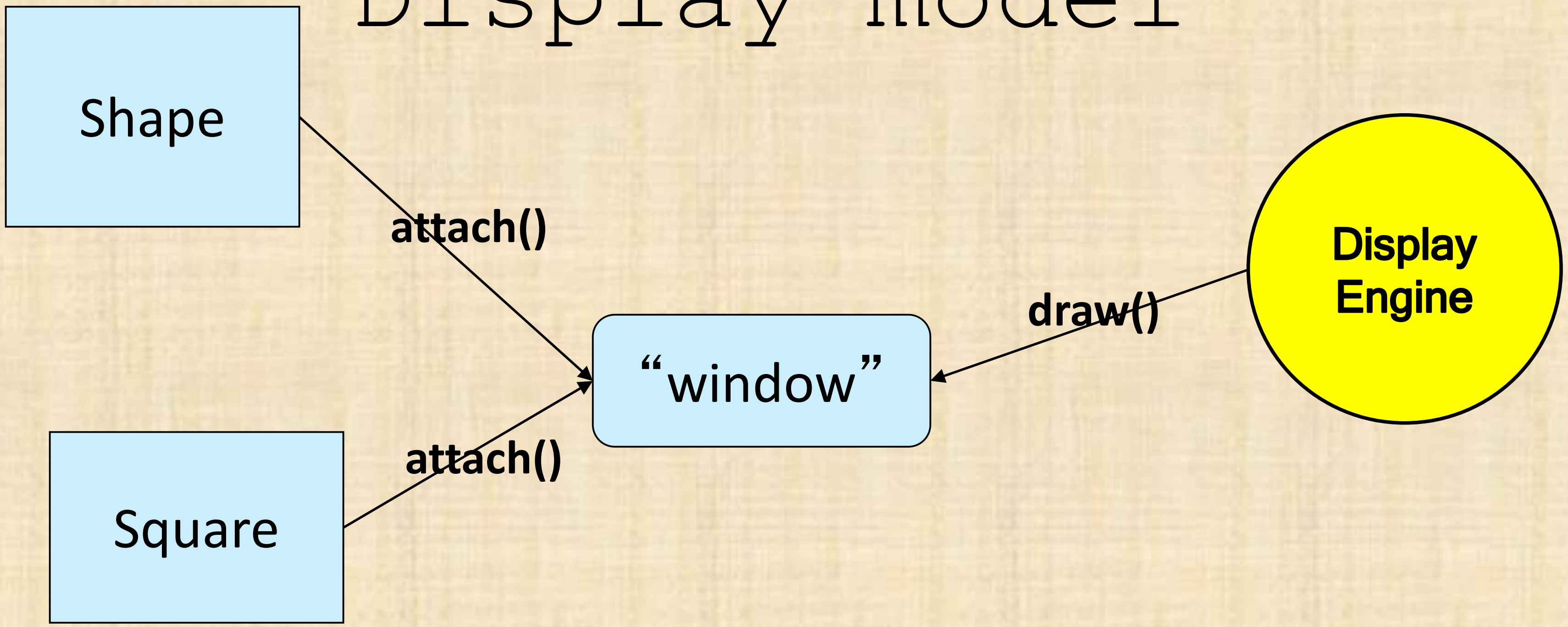
Abstract

- Chapter 10 demonstrated how to create simple windows and display basic shapes: rectangle, circle, triangle, and ellipse. It showed how to manipulate such shapes: change colors and line style, add text, etc.
- Chapter 11 shows how these shapes and operations are implemented and shows a few more examples. In chapter 10, we were basically tool users; here we start to become tool builders.

Overview

- Graphing
  - Model
  - Code organization
- Interface classes
  - Point
  - Line and Lines
  - Grid
  - Polylines
  - Color and Fonts
  - Text
  - Unnamed objects

Display model



- Objects (such as graphs) are "attached to" a window.
- The "display engine" invokes display commands
  (such as "draw line from x to y") for the objects in a window
- Objects such as Square contain vectors of lines, text, etc. for the window to draw

Design note

- The ideal of program design is to represent concepts directly in code
  - We take this ideal very seriously

- For example:
  - **Window** – a window as we see it on the screen
    - Will look different on different operating systems (not our business)
  - **Simple_window –** a window with a “next button”
  - **Line** – a line as you see it in a window on the screen
  - **Point** – a coordinate point
  - **Shape** – what’s common to shapes
    - (imperfectly explained for now; all details in Chapter 12)
  - **Color** – as you see it on the screen

Point

```
namespace Graph_lib {            // our graphics interface is in Graph_lib

    struct Point {               // a Point is simply a pair of ints (the coordinates)
        int x, y;
    };                           // Note the ';'

    // we can compare points:
    bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
    bool operator!=(Point a, Point b) { return !(a==b); }
}
```

Line

```
struct Shape {
   // can hold part of the representation of  a Shape
   // knows how to display Shapes
   //  A Line can be represented (in a Shape) as two Points
};

struct Line : Shape {        // a Line is a Shape defined by just two Points
   Line(Point p1, Point p2) { add(p1);  add(p2); } ;
};
```

Terminology:

- Lines “is derived from” Shape
- Lines “inherits from” Shape
- Lines “is a kind of” Shape
- Shape “is the base” of Lines

This is the key to what is called “object-oriented programming”

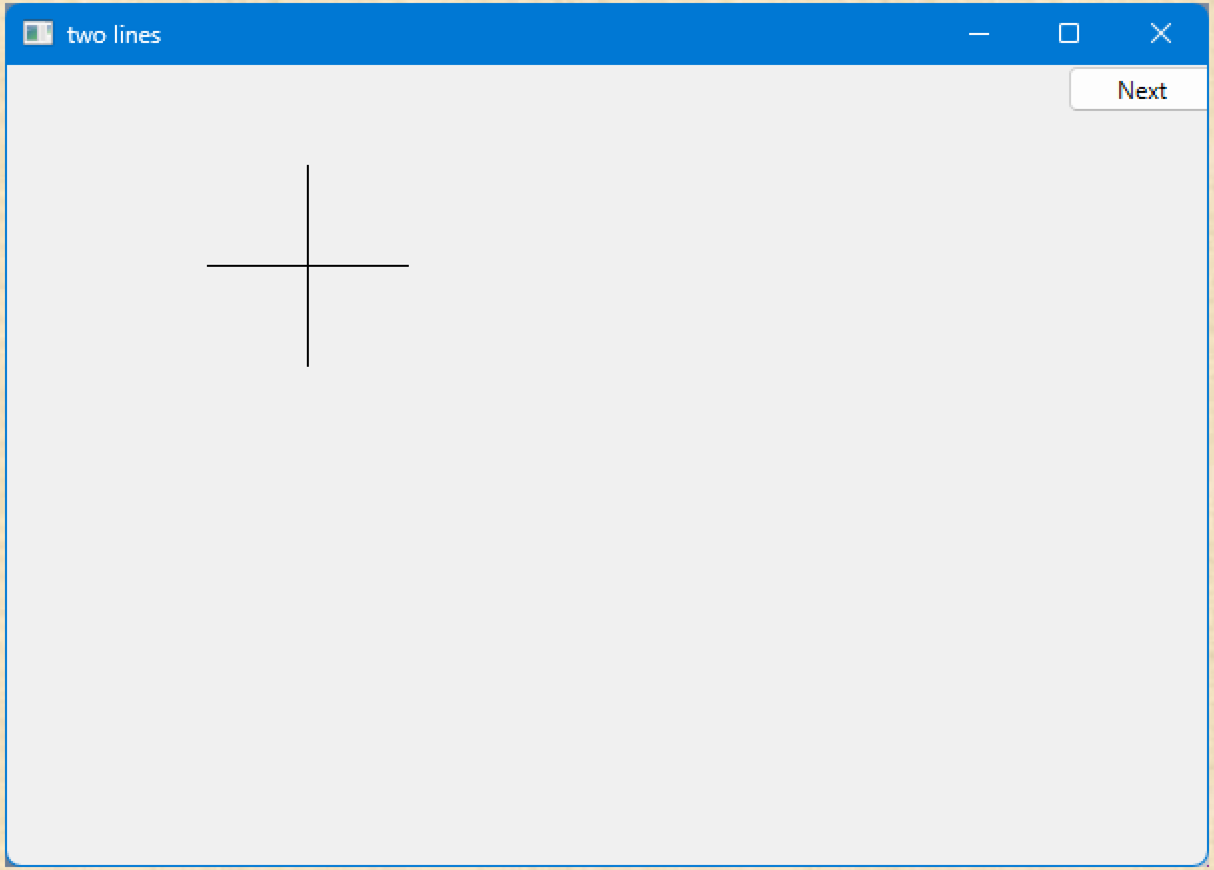
- We’ll get back to this in Chapter 12

Line example

```
// draw two lines:
using namespace Graph_lib;

Simple_window win{Point{100,100},600,400,“Two lines"};          // make a window

Line horizontal {Point {200,100},Point{200,100}};               // make a horizontal line
Line vertical {Point{150,50},Point{150,150}};                   // make a vertical line

win.attach(horizontal);                                         // attach the lines to the window
win.attach(vertical);

win.wait_for_button();                                          // Display!
```

Line example

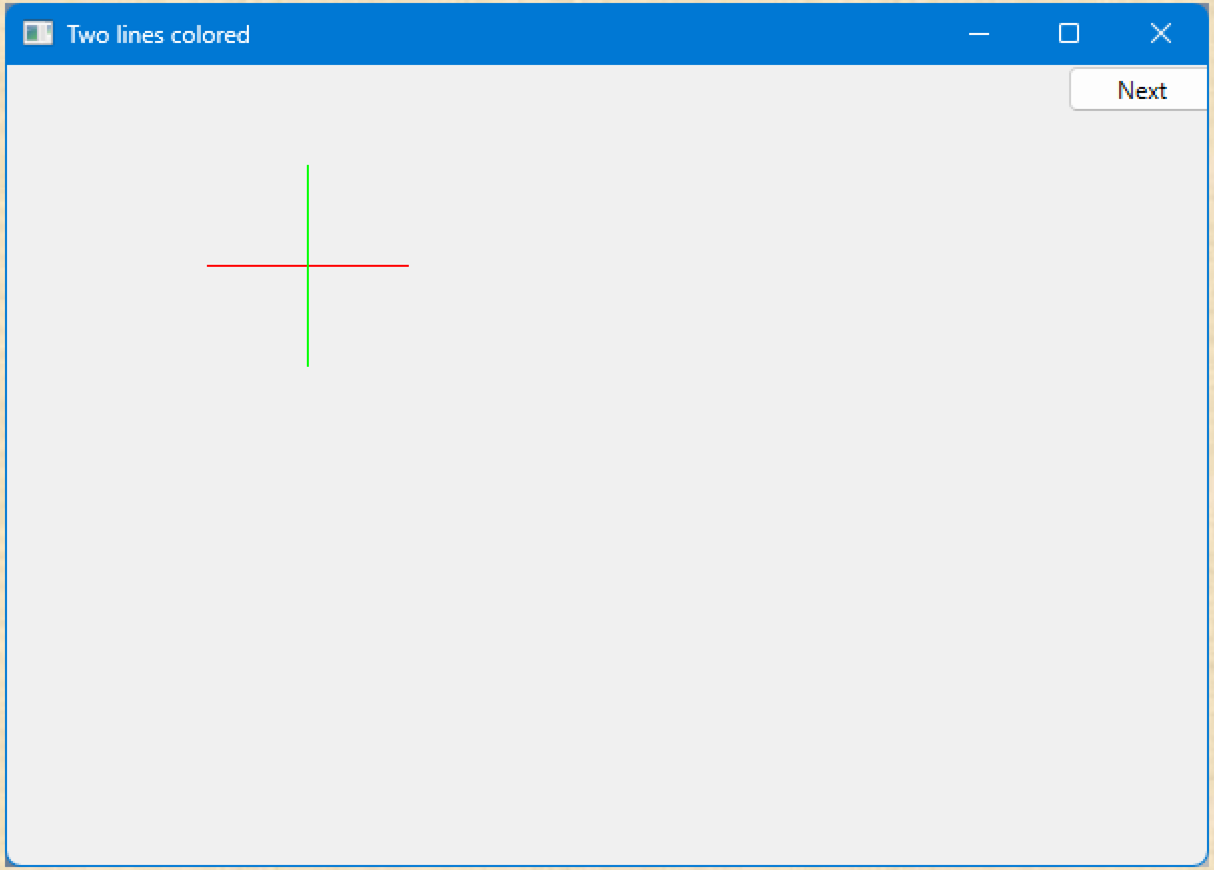
```
using namespace Graph_lib;

Simple_window win{Point{100,100},600,400,“Two lines"};

Line horizontal {Point {200,100},Point{200,100}};
Line vertical {Point{150,50},Point{150,150}};

win.attach(horizontal);
win.attach(vertical);

win.wait_for_button();
```



Line example

- Individual lines are independent

```
horizontal.set_color(Color::red);
vertical.set_color(Color::green);
```



Lines

```
struct Lines : Shape {          // a Lines object is a set of lines
                                // We use Lines when we want to manipulate
                                // all the lines as one shape, e.g., move them all
    Lines(initializer_list<Point> lst = {});        // initialize from a list (possibly empty)

    void add(Point p1, Point p2);       // add line from p1 to p2
protected:
    void draw_specifics(Painter&) const override;
};
```

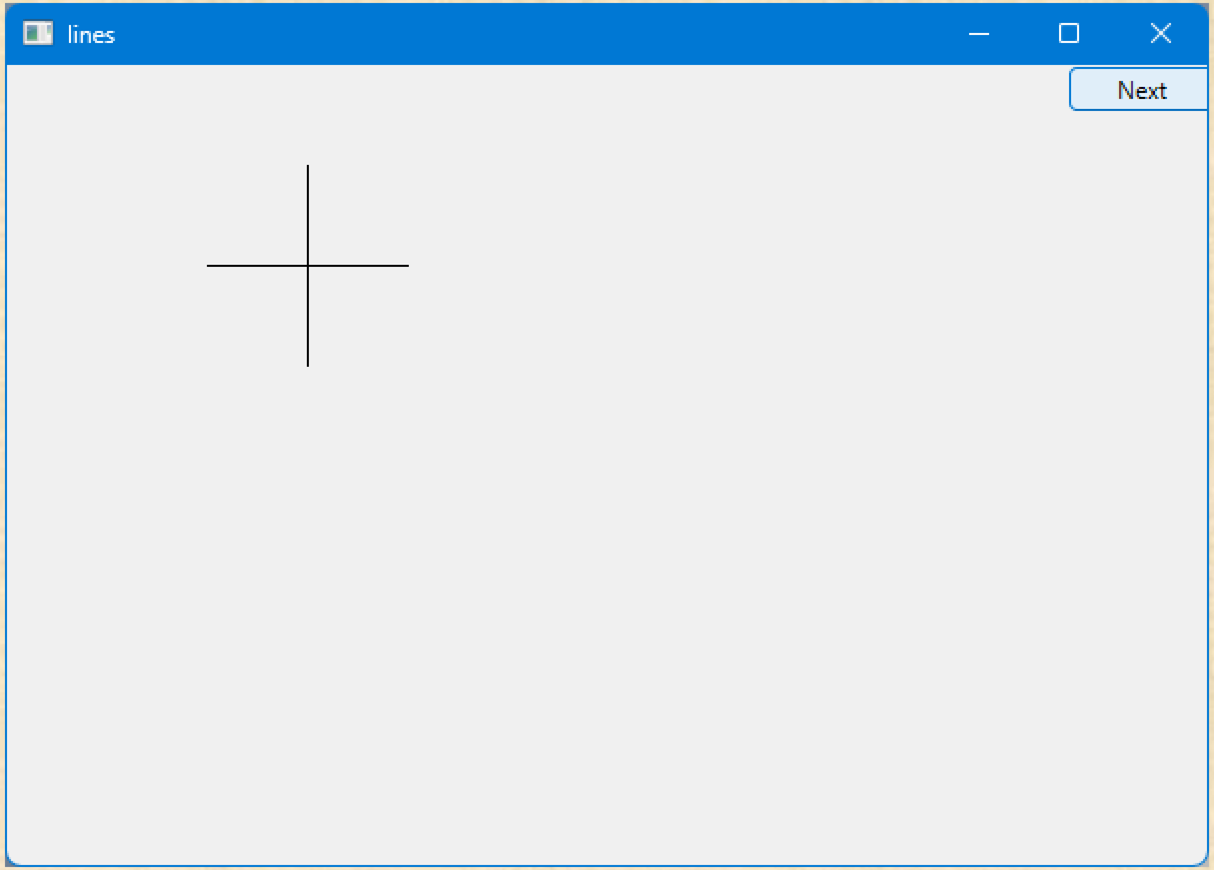
Implementation details

- **draw_specifics()** is to be used only by parts of the Lines implementation. Making it **protected** ensures that.
- **Painter** is part of the interface to our underlying Qt library. Never used directly by users.
- **override** says that **draw_specifics()** is to be used instead of **Shape**'s own **draw_specifics()**.

Lines Example

```
Lines x = {
    {Point{100,100}, Point{200,100}},            // first line: horizontal
    {Point{150,50}, Point{150,150}}  // second line: vertical
};
```

- It looks *exactly* like the two **Lines** example




```
// or even terser this:
Lines x = {
    {{100,100}, {200,100}},   {{150,50}, {150,150}}
};

// but don't overdo abbreviation/terseness
// code is meant to be read
```

Implementation: Lines

```
Lines::Lines(std::initializer_list<Point> lst)
    : Shape{lst}
{
    if (lst.size() % 2)
            error("odd number of points for Lines");
}

void Lines::add(Point p1, Point p2)        // use Shape's add()
{
    Shape::add(p1);
    Shape::add(p2);
    redraw();                  // we have changed the Lines object; let's see it
}
```

Implementation: Lines

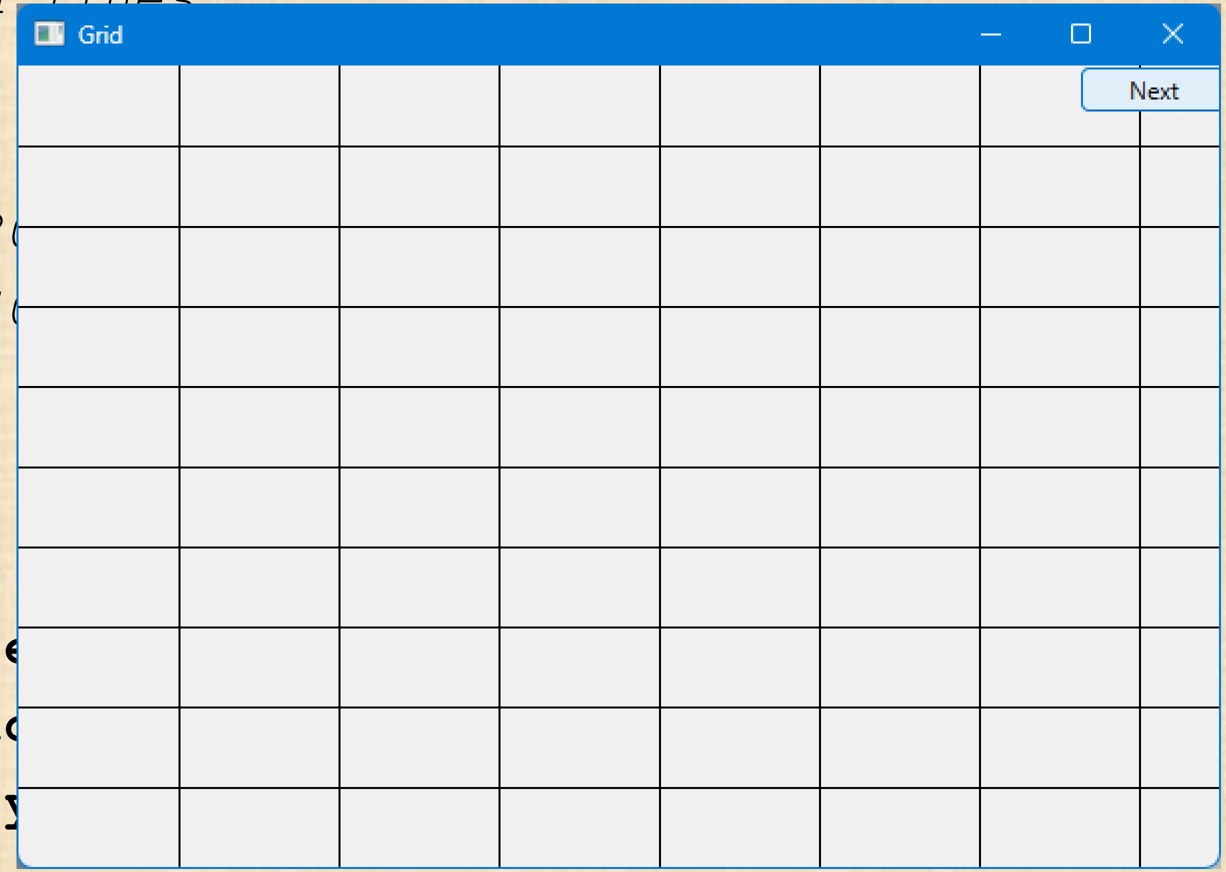
```
void Lines::draw_specifics(Painter& painter) const
{
   if (color().visibility())
            for (int i=1; i<number_of_points(); i+=2)
                        painter.draw_line(point(i-1) ,point(i));
}
```

- Note
  - **painter.draw_line()** is a basic line drawing function from Qt
  - Qt is used in the *implementation*, not in the *interface* to our classes
  - We could replace Qt with another graphics library; in fact, Qt did replace another library

Draw a grid

(Why bother with **Lines** when we have **Line**?)

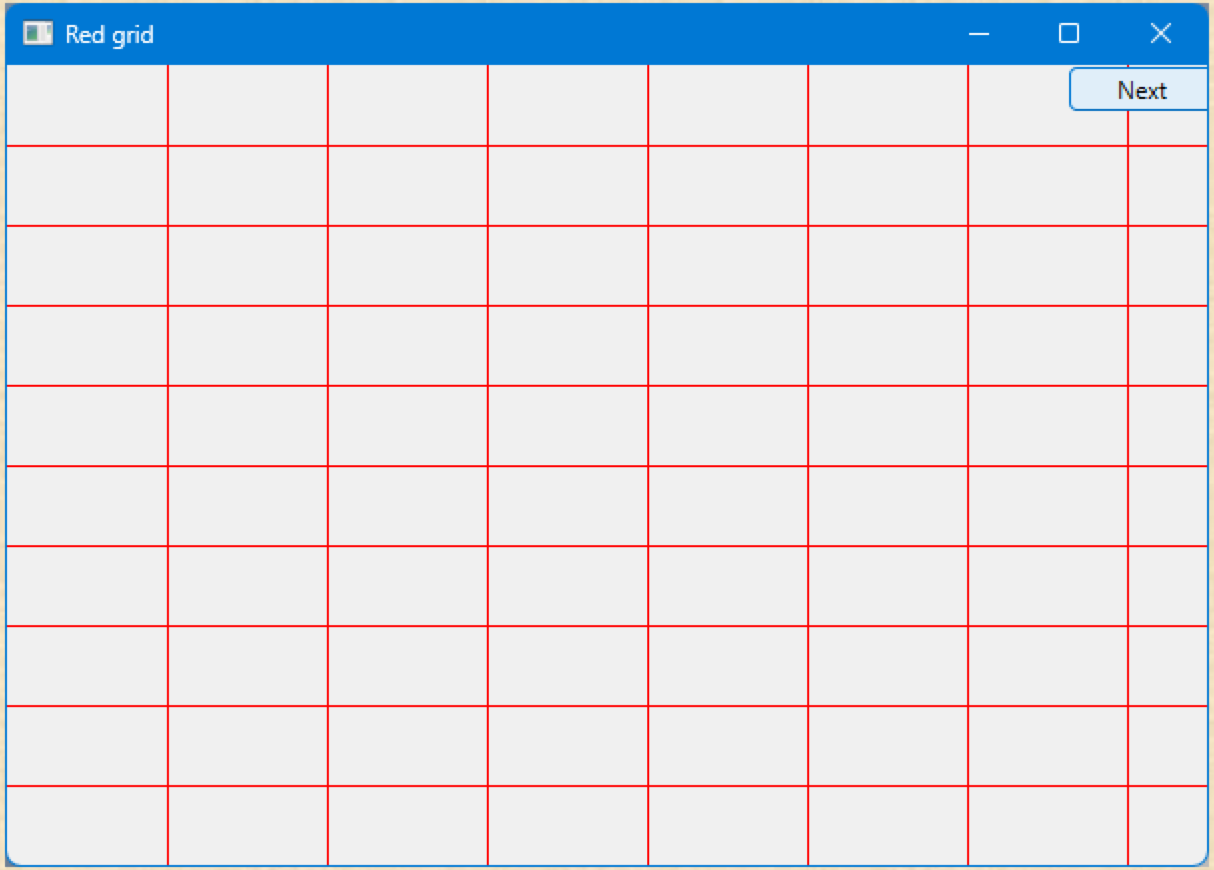
```
// A Lines object may hold many related lines
int x_size = win.x_max();
int y_size = win.y_max();
int x_grid = 80;              // make cells 8
int y_grid = 40;              // make cells 4

Lines grid;

for (int x=x_grid; x<x_size; x+=x_grid)
      grid.add(Point(x,0),Point(x,y_size
for (int y = y_grid; y<y_size; y+=y_gric
      grid.add(Point(0,y),Point(x_size,y

win.attach(grid);             // attach our grid to our window (note:
  grid is one object)
```



Draw red grid

That grid was a bit pale, let's add color

```
grid.set_color(Color::red);
```



Color

```
struct Color {     // Map Qt colors and scope them; deal with
  visibility/transparency
  enum Color_type {
      red, blue, green, yellow, white, black, magenta, cyan, dark_red,
      // named colors
      dark_green, dark_yellow, dark_blue, dark_magenta, dark_cyan,
      palette_index,          // refers to a set of popular colors
      rgb                     // refers to the usual red-green-blue
  representation of color
  };
  enum Transparency { invisible = 0, visible=255 };    // control of
  visibility
  // … constructors and access functions …
private:
  int c = 0;
  Color_type ct = black;
  struct Rgb { int r; int g; int b; };
  Rgb rgb_color = {0,0,0};
```

Color

```
struct Color {
	// …
	Color(Color_type cc) :c{cc}, ct{cc}, v{visible} { }			// use named colors
	Color(Color_type cc, Transparency vv) :c{cc}, ct{cc}, v{vv} { }
	Color(int cc) :c{cc}, ct{Color_type::palette_index}, v{visible} { }
	// choose from palette
	Color(Transparency vv) :c{}, ct{Color_type::black}, v{vv} { }
	Color(int r, int g, int b) :c{}, ct{Color_type::rgb}, rgb_color{r,g,b}, v{visible} {}	// use RGB

	int as_int() const { return c; }
	int red_component() const { return rgb_color.r; }
	int green_component() const { return rgb_color.g; }
	int blue_component() const { return rgb_color.b; }
	Color_type type() const { return ct; }

	char visibility() const { return v; }
	void set_visibility(Transparency vv) { v=vv; }
```

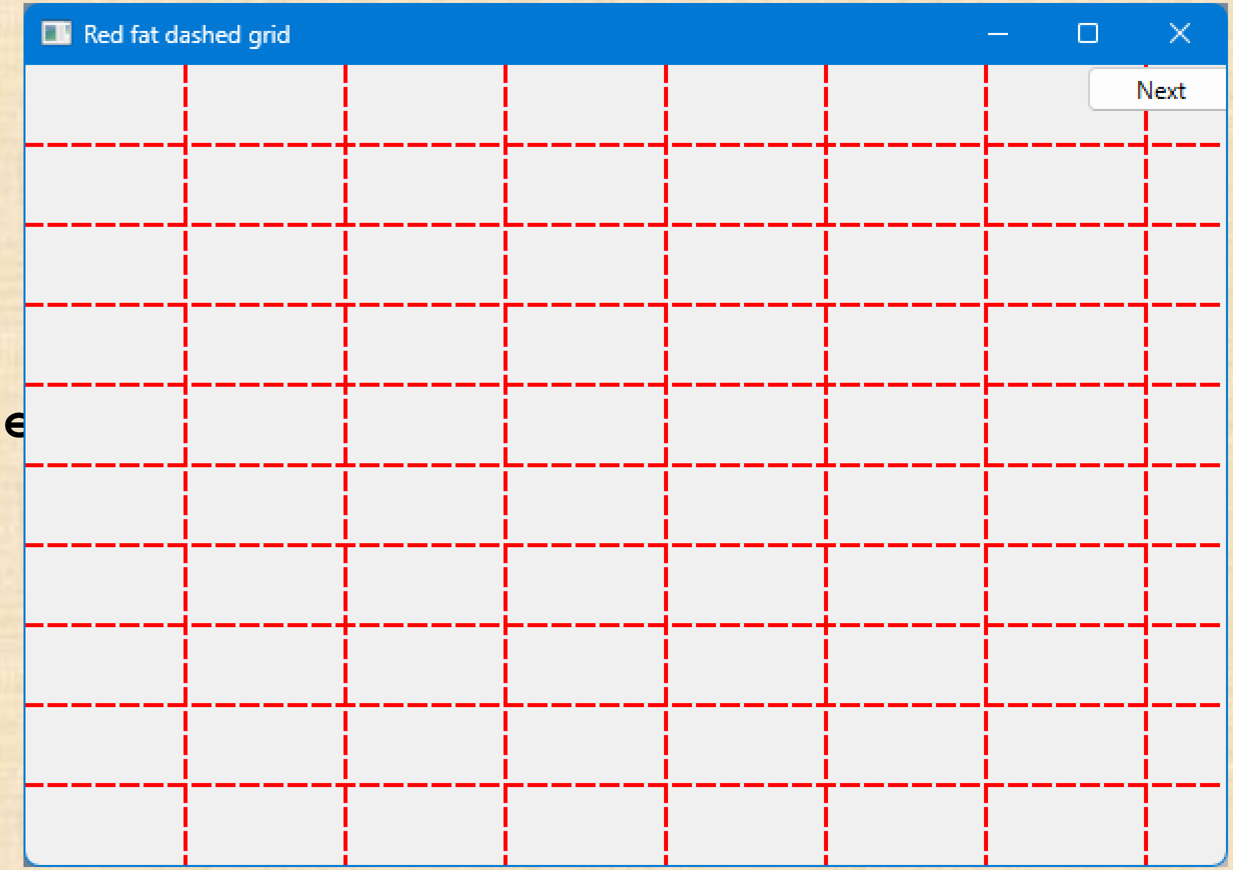
Example: colored fat dash grid

That grid is a bit thin,

and maybe we prefer dashed lines

```
grid.set_style(Line_style{Line_style
```

- Line styles are named
- Line thickness are measures in pixels

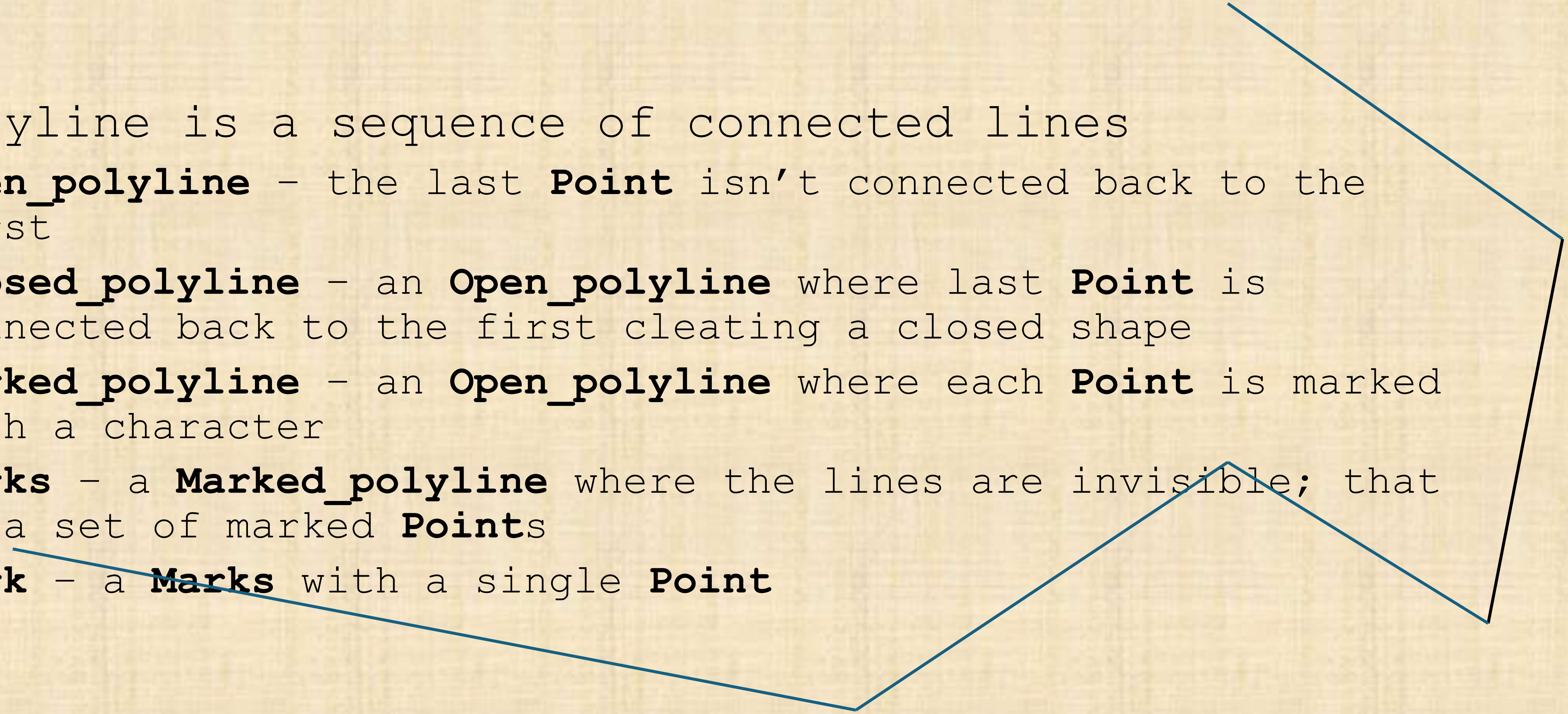


Line_style

```
struct Line_style {
  enum Line_style_type {
        solid,                          // -------
        dash,                           // - - - -
        dot,                            // .......
        dashdot,                        // - . - .
        dashdotdot                      // -..-..
  };
  Line_style(Line_style_type ss) :s{ss} { }
  Line_style(Line_style_type ss, int ww) :s{ss}, w(ww) { }
  Line_style() {}

  int width() const { return w; }
  int style() const { return s; }
private:
  int s = solid;
  int w = 1;
};
```

Polylines

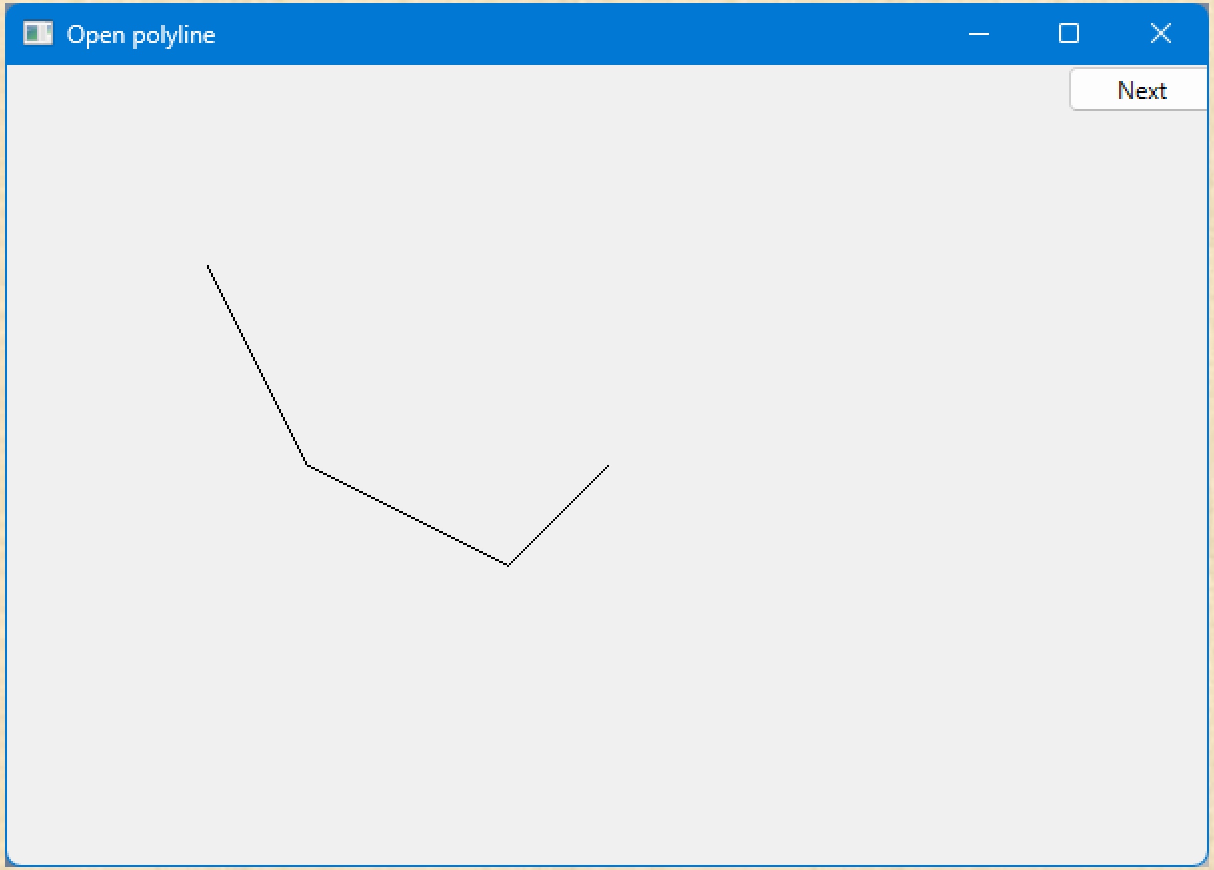
- A polyline is a sequence of connected lines
  - **Open_polyline** – the last **Point** isn’t connected back to the first
  - **Closed_polyline** – an **Open_polyline** where last **Point** is connected back to the first cleating a closed shape
  - **Marked_polyline** – an **Open_polyline** where each **Point** is marked with a character
  - **Marks** – a **Marked_polyline** where the lines are invisible; that is a set of marked **Point**s
  - **Mark** – a **Marks** with a single **Point**



Open_polyline

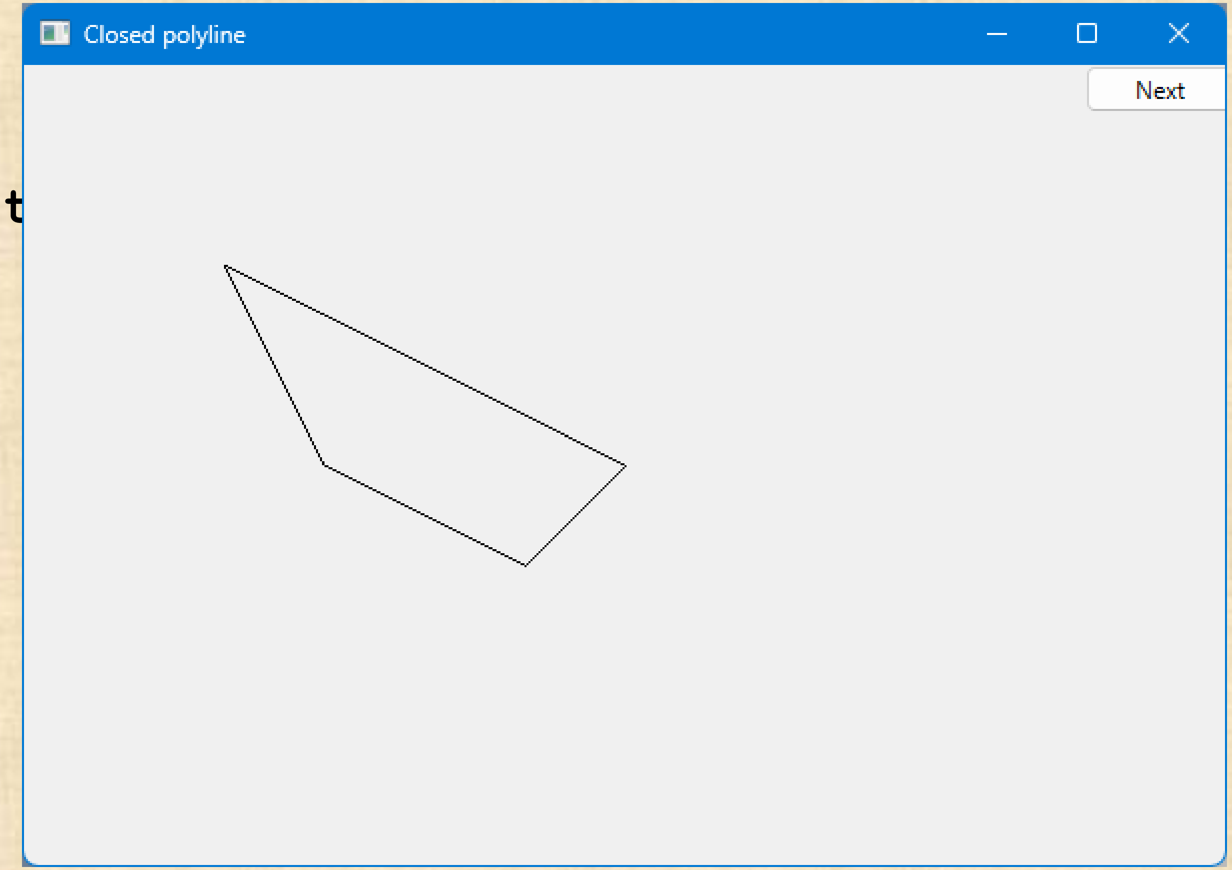
```
struct Open_polyline : Shape {        // open sequence of lines
    Open_polyline(std::initializer_list<Point> p = {}) : Shape(p) {}
    void add(Point p) { Shape::add(p); redraw(); }
protected:
    void draw_specifics(Painter&) const override;
};
```

```
Open_polyline opl = {
    {100,100},
    {150,200},
    {250,250},
    {300,200}
};
```



Closed_polyline

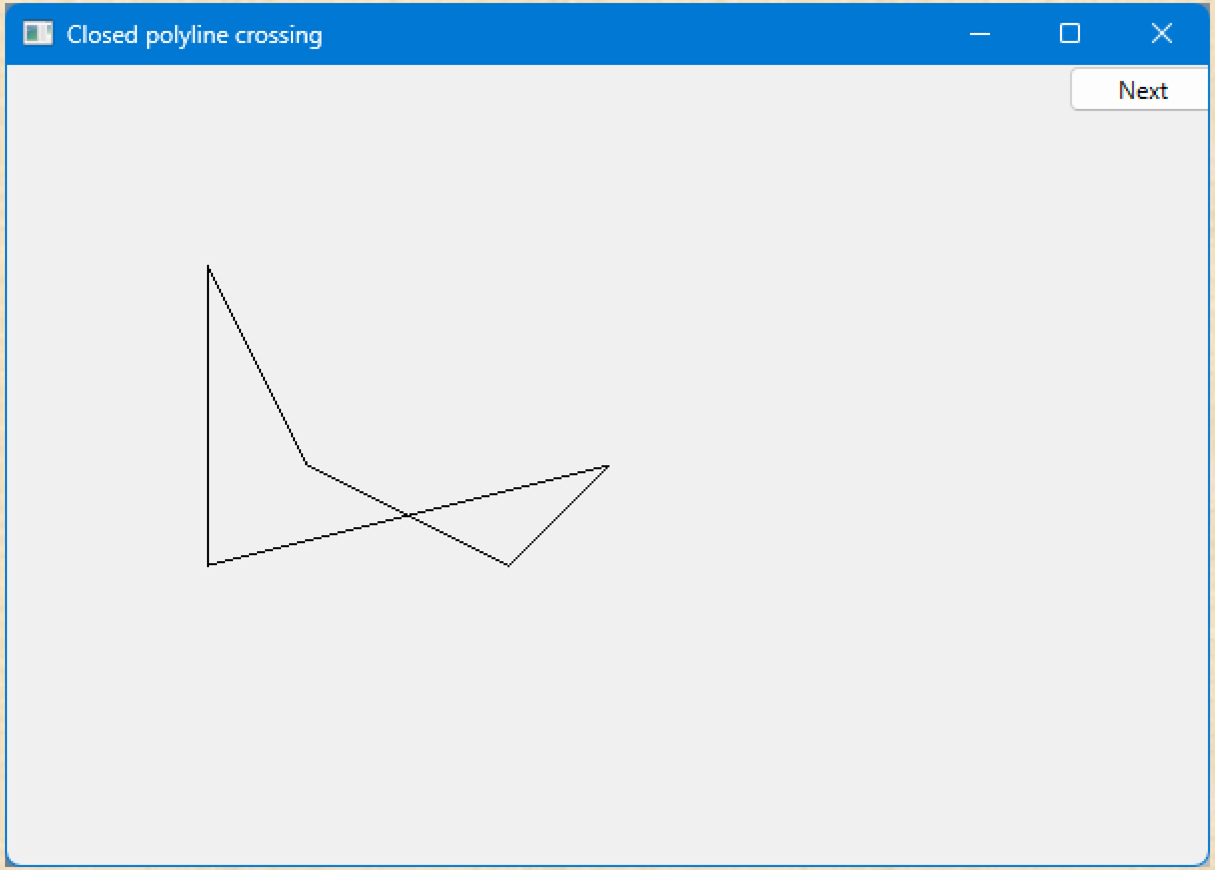
```
struct Closed_polyline : Open_polyline {   // closed sequence of lines
  using Open_polyline::Open_polyline;
protected:
    void draw_specifics(Painter&) const
};

Closed_polyline cpl = {
  {100,100},
  {150,200},
  {250,250},
  {300,200}
};
```



Closed_polyline

- A **Closed_polyline** is not a polygon
  - some **Closed_polyline**s look like polygons
- A **Polygon** is a **Closed_polyline**
  - where no lines cross
  - A **Polygon** has a stronger invariant than a **Closed_polyline**

```
cpl.add(Point{100,250});
```

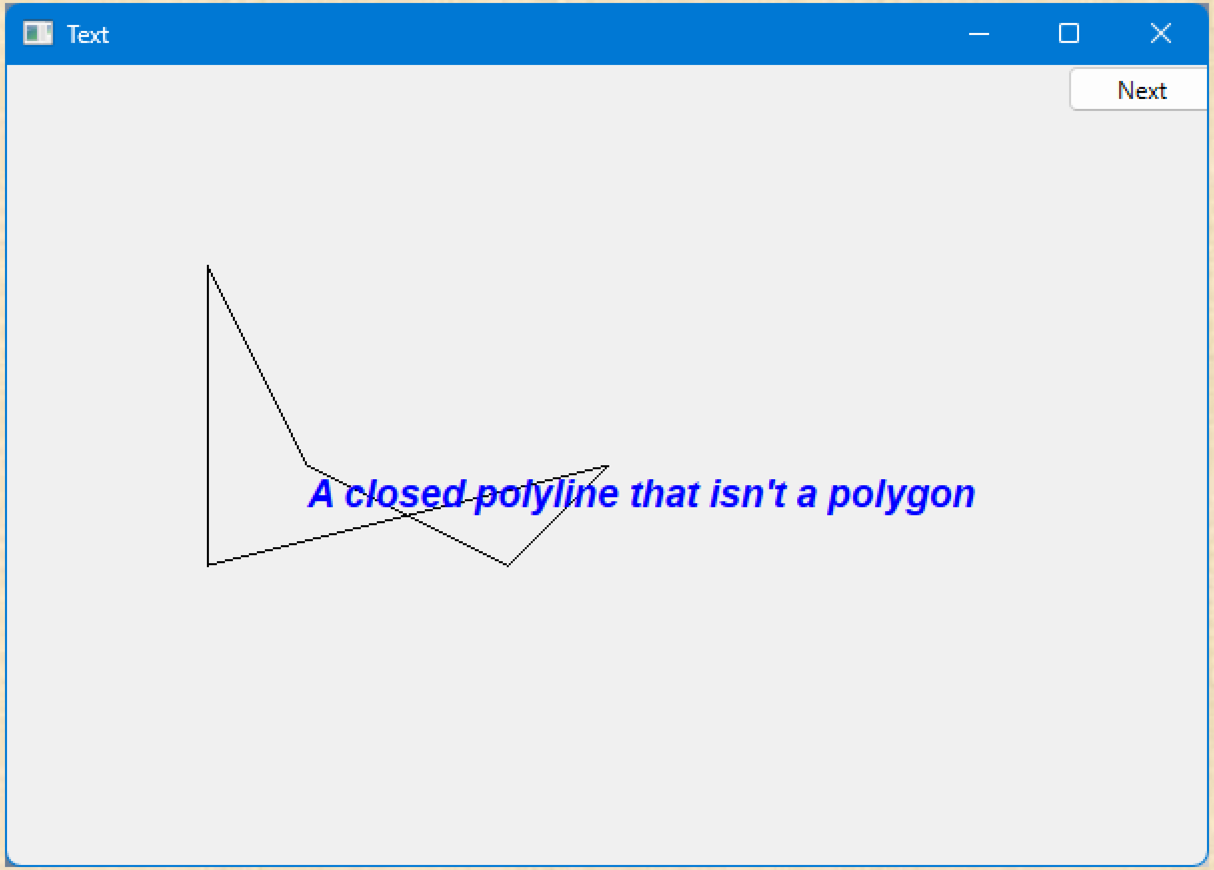


Text

```
struct Text : Shape {
  Text(Point x, const string& s) : lab{ s } { add(x); }      // the point is the top left of the first letter

    void set_label(const string& s) { lab = s; redraw();}   // a text is of a color
  string label() const { return lab; }

    void set_font(Font f) { fnt = f; redraw();}        // a text uses a specific font
  Font font() const { return Font(fnt); }

    void set_font_size(int s) { fnt_sz = s; redraw();}           // the characters of a text has a size
  int font_size() const { return fnt_sz; }
protected:
    void draw_specifics(Painter&) const override;
private:
  string lab;            // label; that is the text string
    Font fnt = Font::courier;
```

Add text

```
Text t {Point{150,200}, "A closed polyline that isn’t a
  polygon"};
t.set_color(Color::blue);
t.set_font(Font::Helvetica_bold_italic)
```

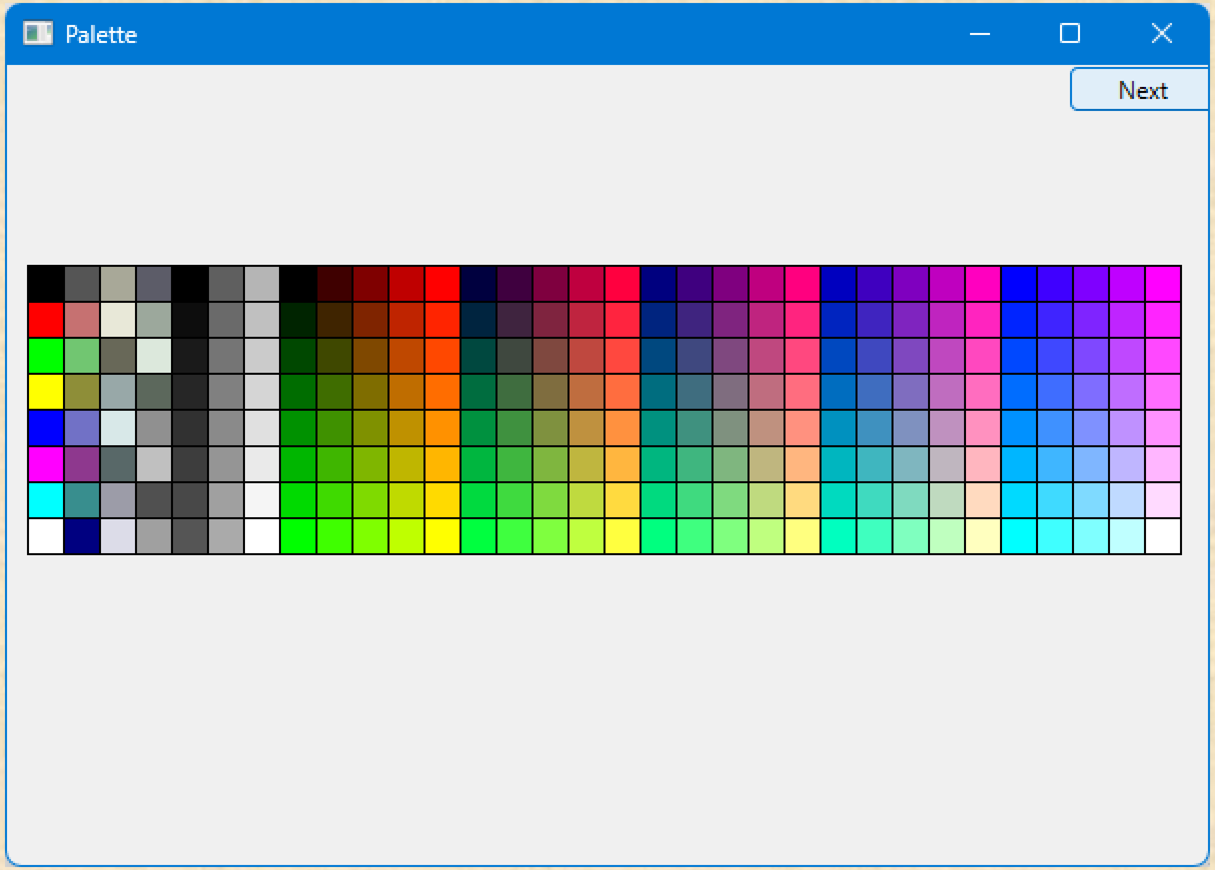


Font

```
struct Font {
        enum Font_type {
                helvetica, helvetica_bold, helvetica_italic, 
helvetica_bold_italic,
                courier, courier_bold, courier_italic, 
courier_bold_italic,
                times, times_bold, times_italic, times_bold_italic,
                symbol,
                screen, screen_bold,
                zapf_dingbats
        };
.       Font(Font_type ff) :f(ff) { }
        int as_int() const { return f; }
private:
        int f = courier;
```

Color matrix (32*8)

- Let's draw a color matrix
- To see
  - some of the colors we have to work with
  - how messy two-dimensional addressing can be
    - See PPP2 Chapter 24 for real matrices
  - how to avoid inventing names for objects

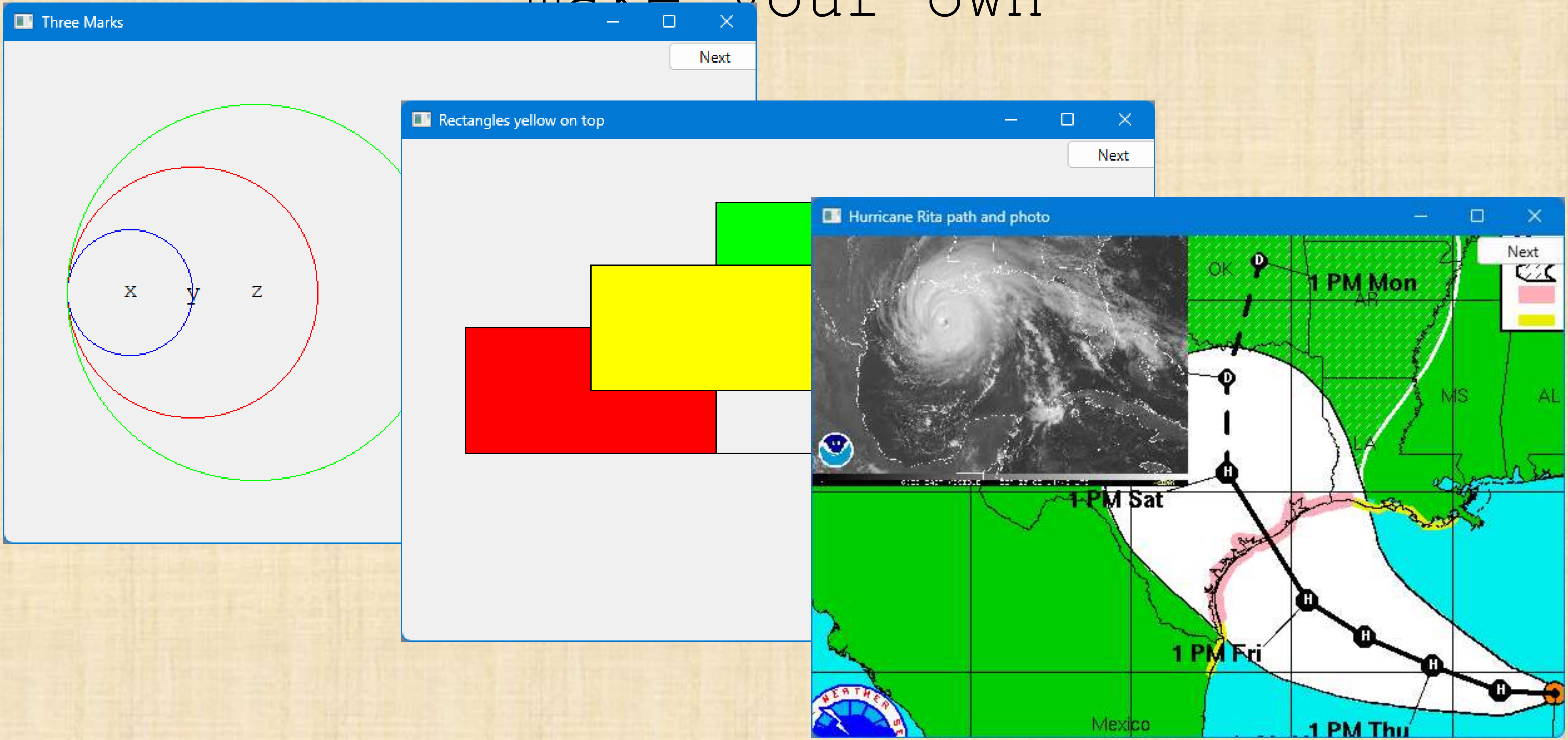


Color matrix (32*8)

```
Vector_ref<Rectangle> vr;       // use like a vector and imagine that it
holds Rectangle& elements

const int max = 32;             //number of columns
const int side = 18;            // size of color rectangle
const int left = 10;            // left edge
const int top = 100;            // top edge
int color_index = 0;

for (int i = 0; i < max; ++i) {                   // all columns
      for (int j = 0; j < 8; ++j) {               // 8 rows in each column
            vr.push_back(make_unique<Rectangle>(Point{
i*side+left,j*side+top }, side, side));
            vr[vr.size()-1].set_fill_color(color_index);
            ++color_index;                  // move to the next color
            win.attach(vr[vr.size()-1]);
      }
```

Make an unnamed Rectangl
(details in Chapter 18)

There are more Shapes – and you can make your own



Next lecture

- What is class **Shape**?
- Introduction to object-oriented programming