

Chapter 13

Graphics classes

Bjarne Stroustrup

www.stroustrup.com/Programming

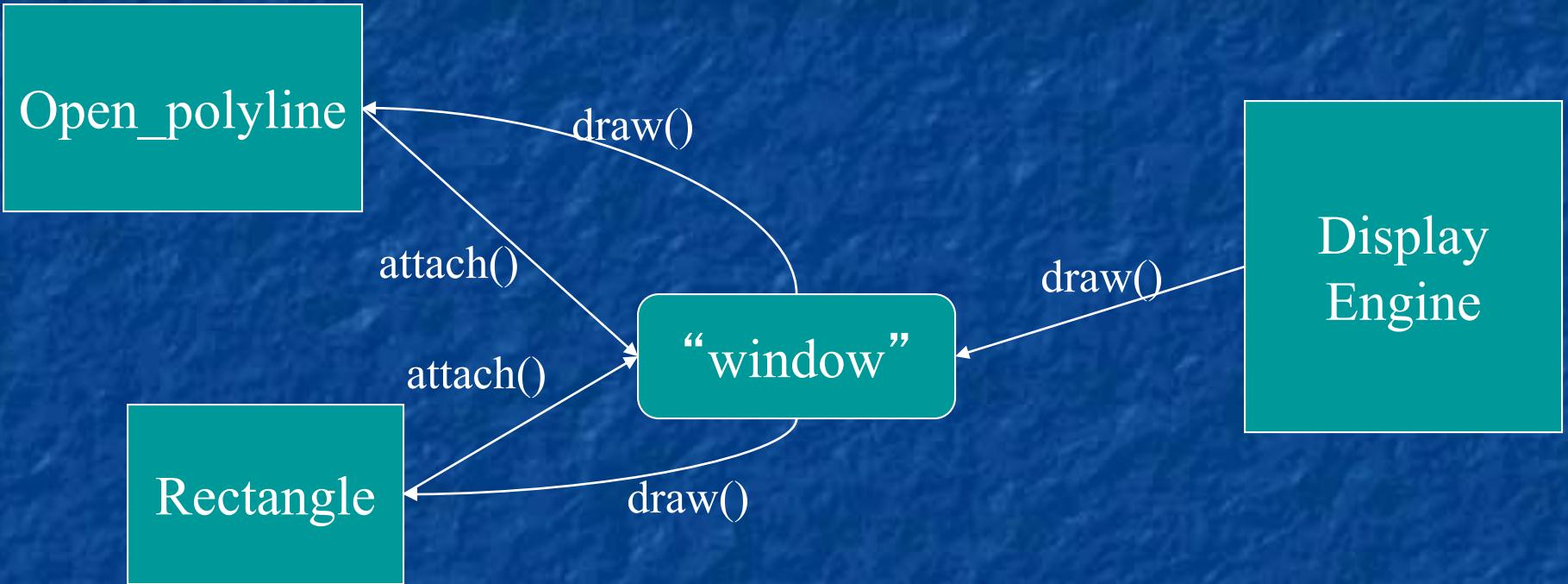
Abstract

- Chapter 12 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 13 shows how these shapes and operations are implemented, and shows a few more examples. In Chapter 12, we were basically tool users; here we become tool builders.

Overview

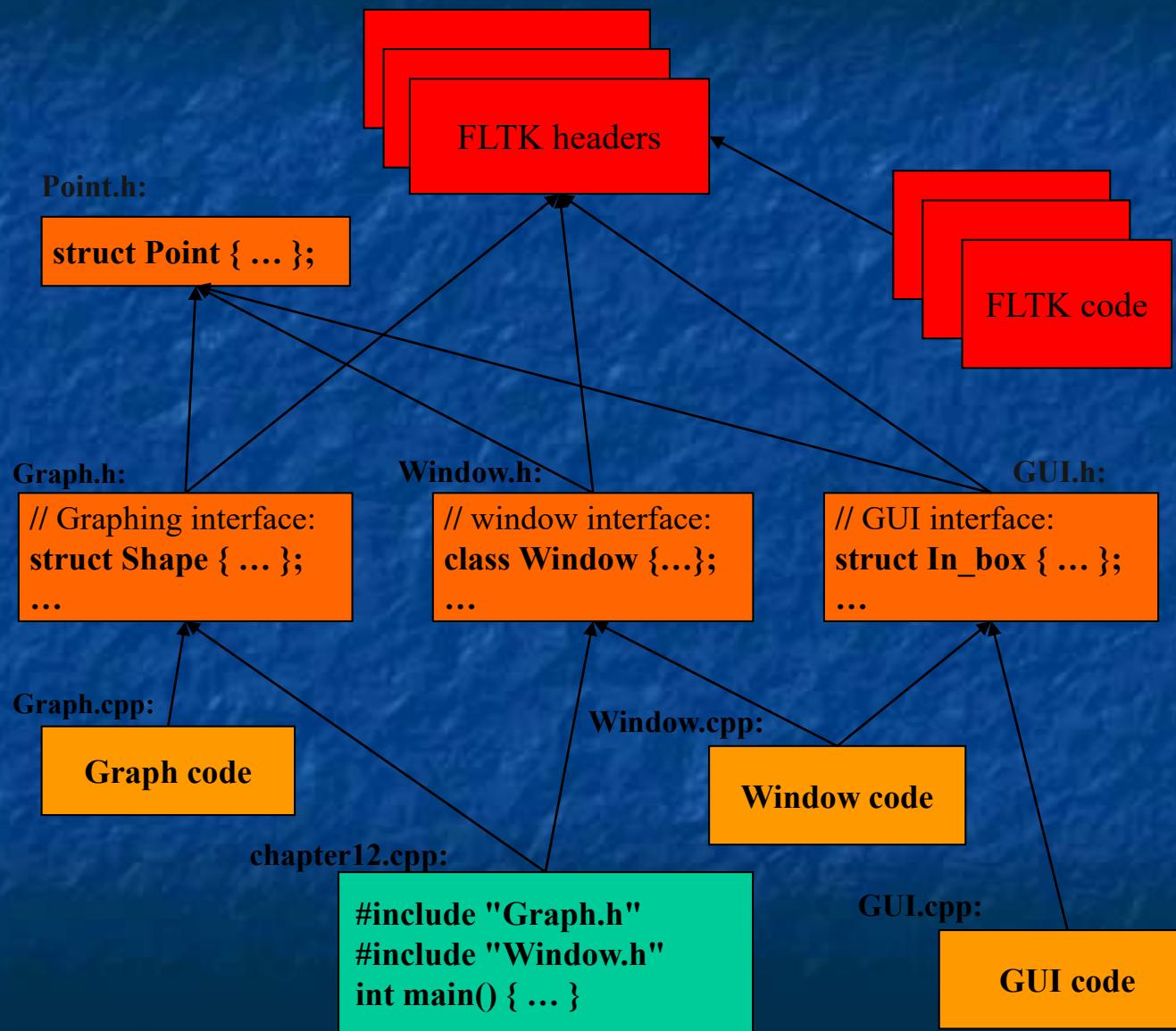
- Graphing
 - Model
 - Code organization
- Interface classes
 - Point
 - Line
 - Lines
 - Grid
 - Open Polylines
 - Closed Polylines
 - Color
 - Text
 - Unnamed objects

Display model



- Objects (such as graphs) are “attached to” (“placed in”) 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 Rectangle add vectors of lines to the window to draw

Code organization



Source files

- Header
 - File that contains interface information (declarations)
 - **#include** in user and implementer
- .cpp (“code file” / “implementation file”)
 - File that contains code implementing interfaces defined in headers and/or uses such interfaces
 - **#includes** headers
- Read the **Graph.h** header
 - And later the **Graph.cpp** implementation file
- Don’t read the **Window.h** header or the **Window.cpp** implementation file
 - Naturally, some of you will take a peek
 - Beware: heavy use of yet unexplained C++ features

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)
 - **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 14)
 - **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;
        Point(int xx, int yy) : x(xx), y(yy) { }
    };
}
```

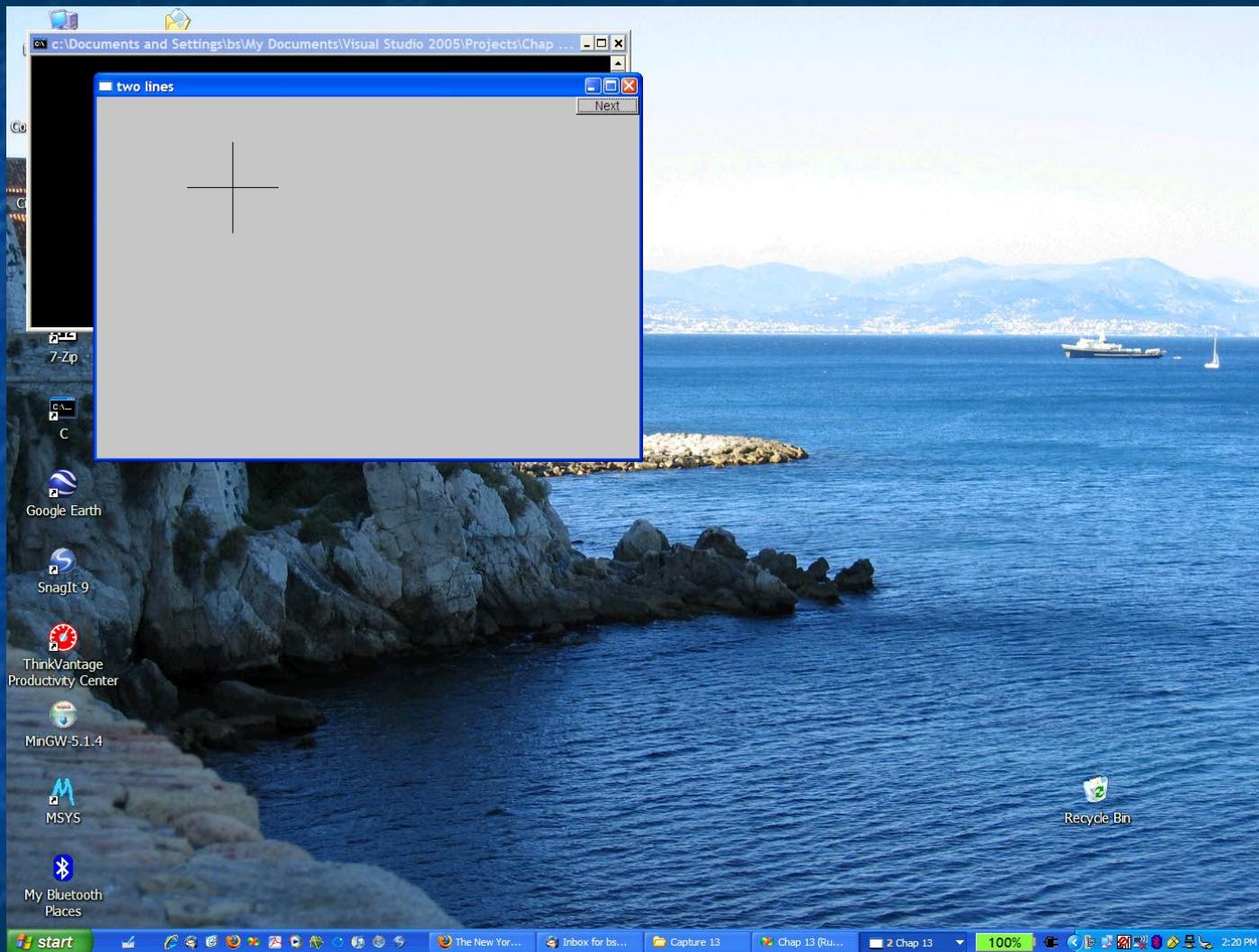
Line

```
struct Shape {  
    // hold lines represented as pairs of points  
    // knows how to display lines  
};  
  
struct Line : Shape          // a Line is a Shape defined by just two Points  
{  
    Line(Point p1, Point p2);  
};  
  
Line::Line(Point p1, Point p2)    // construct a line from p1 to p2  
{  
    add(p1);      // add p1 to this shape (add() is provided by Shape)  
    add(p2);      // add p2 to this shape  
}
```

Line example

```
// draw two lines:  
using namespace Graph_lib;  
  
Simple_window win(Point(100,100),600,400,"Canvas"); // make a window  
  
Line horizontal(Point(100,100),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!
```

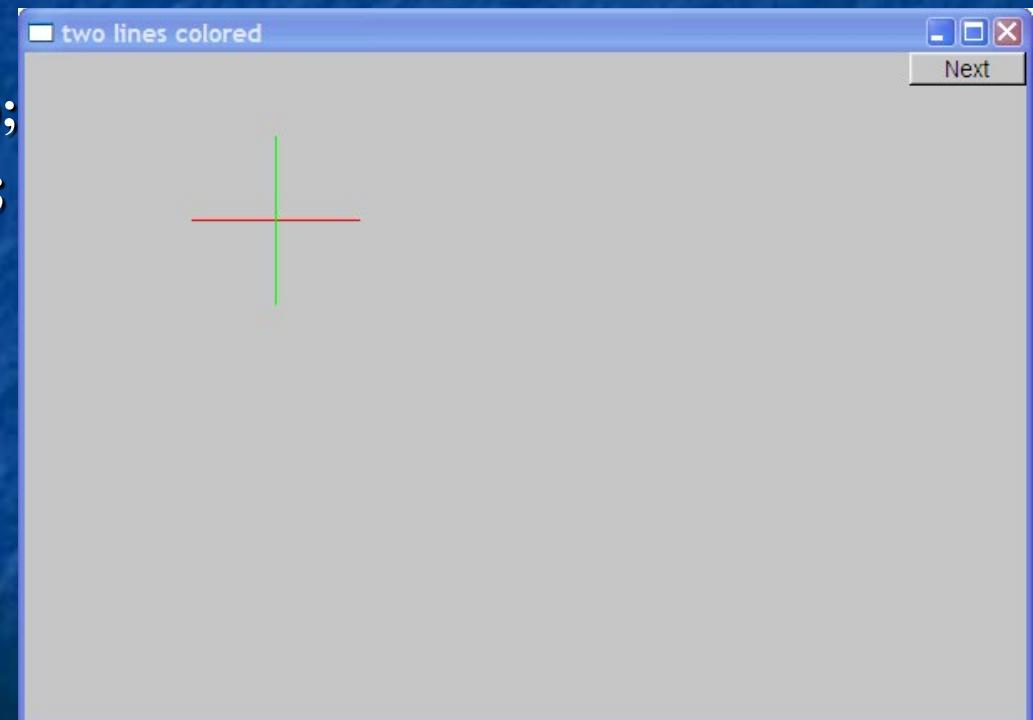
Line example



Line example

- Individual lines are independent

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



Lines

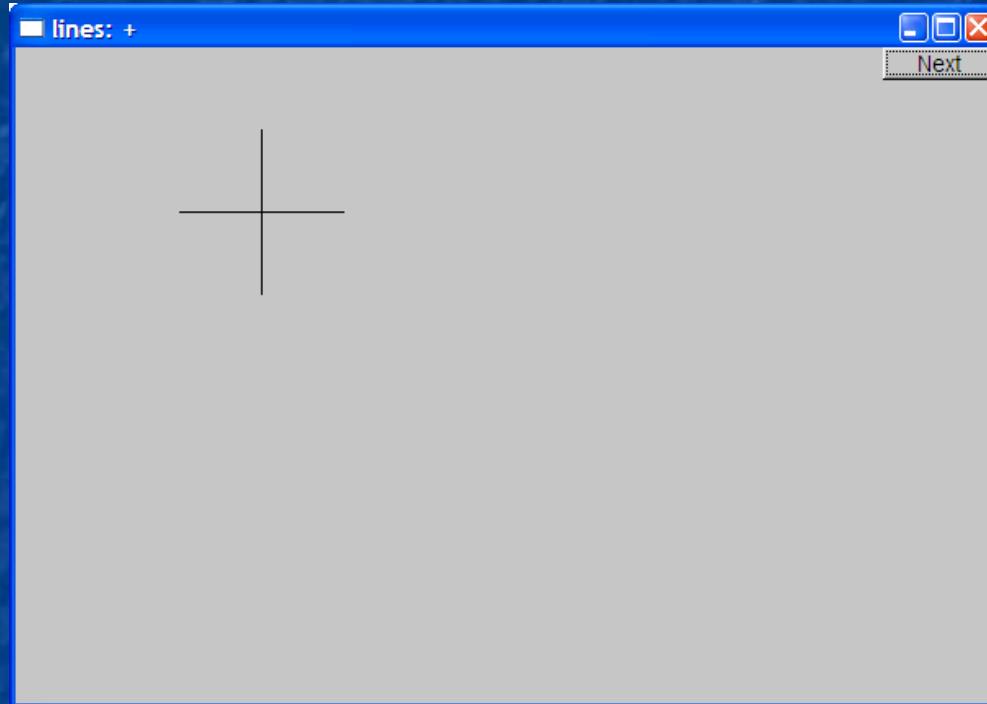
```
struct Lines : Shape {    // a Lines object is a set of lines
                        // We use Lines when we want to manipulate
                        // all the lines as one shape, e.g. move them all
                        // together with one move statement
    void add(Point p1, Point p2); // add line from p1 to p2
    void draw_lines() const;    // to be called by Window to draw Lines
};
```

- 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 14

Lines Example

```
Lines x;  
x.add(Point(100,100), Point(200,100));      // horizontal line  
x.add(Point(150,50), Point(150,150));      // vertical line  
  
win.attach(x);          // attach Lines object x to Window win  
win.wait_for_button(); // Draw!
```

Lines example



- Looks exactly like the two Lines example

Implementation: Lines

```
void Lines::add(Point p1, Point p2)          // use Shape's add()
{
    Shape::add(p1);
    Shape::add(p2);
}

void Lines::draw_lines() const    // to somehow be called from Shape
{
    for (int i=1; i<number_of_points(); i+=2)
        fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

■ Note

- `fl_line` is a basic line drawing function from FLTK
- FLTK is used in the *implementation*, not in the *interface* to our classes
- We could replace FLTK with another graphics library

Draw Grid

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

// A **Lines** object may hold many related lines

// Here we construct a grid:

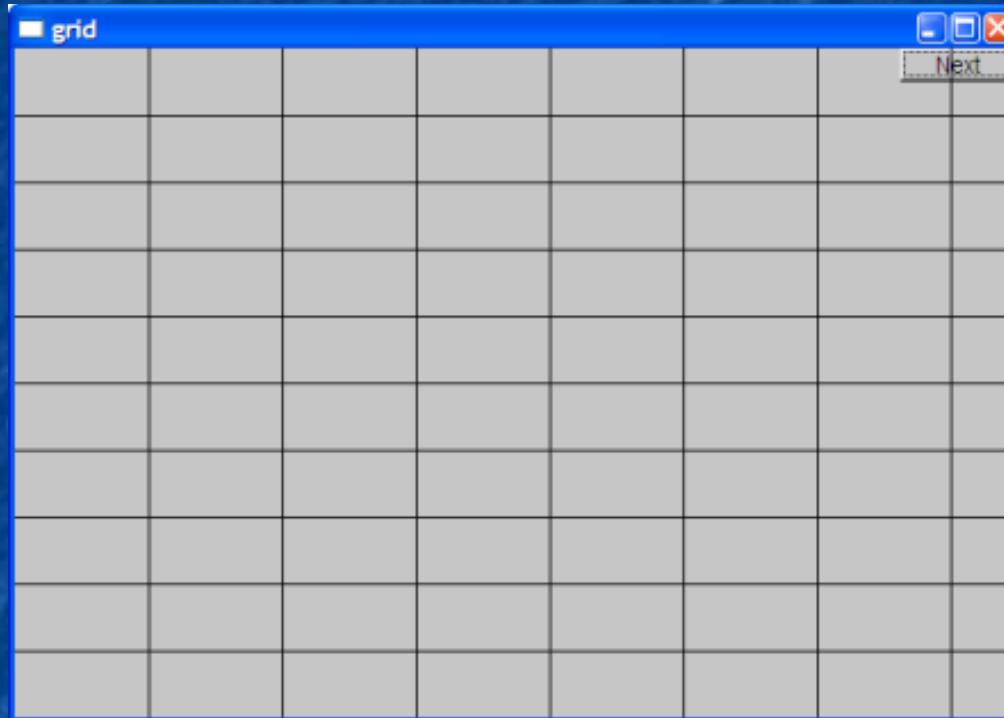
```
int x_size = win.x_max();  
int y_size = win.y_max();  
int x_grid = 80;           // make cells 80 pixels wide  
int y_grid = 40;           // make cells 40 pixels high
```

Lines grid;

```
for (int x=x_grid; x<x_size; x+=x_grid)    // vertical lines  
    grid.add(Point(x,0),Point(x,y_size));  
for (int y = y_grid; y<y_size; y+=y_grid)    // horizontal lines  
    grid.add(Point(0,y),Point(x_size,y));
```

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

Grid



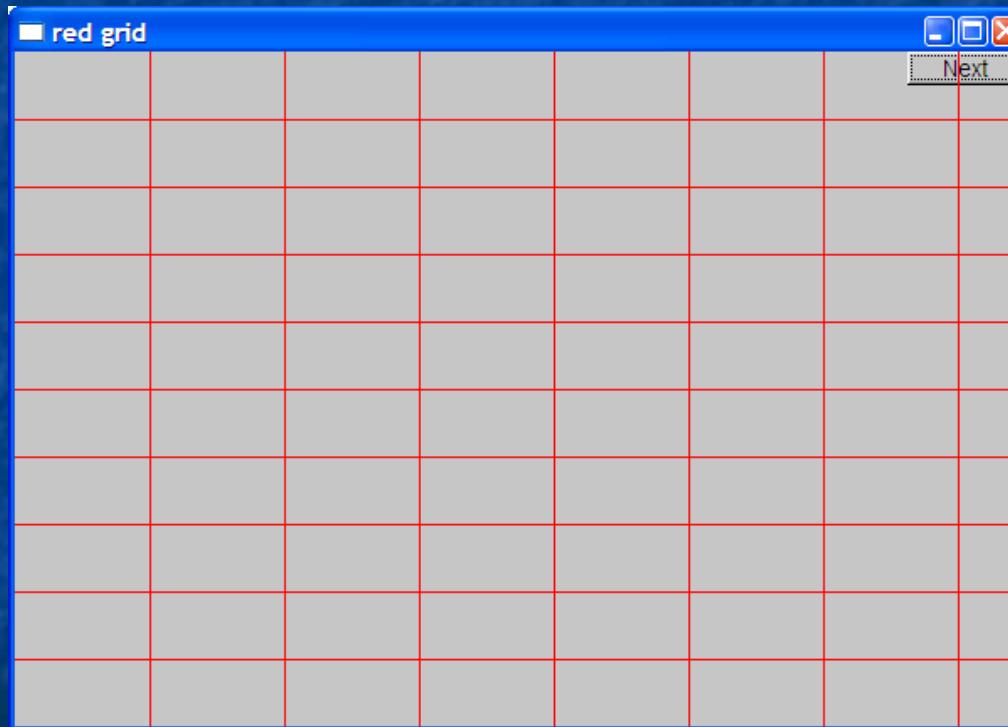
- Oops! Last column is narrow, there's a grid line on top of the Next button, etc.—tweaking required (as usual)

Color

```
struct Color {    // Map FLTK colors and scope them;  
    // deal with visibility/transparency  
    enum Color_type { red=FL_RED, blue=FL_BLUE, /* ... */ };  
  
    enum Transparency { invisible=0, visible=255 }; // also called Alpha  
  
    Color(Color_type cc) :c(Fl_Color(cc)), v(visible) {}  
    Color(int cc) :c(Fl_Color(cc)), v(visible) {}  
    Color(Color_type cc, Transparency t) :c(Fl_Color(cc)), v(t) {}  
  
    int as_int() const { return c; }  
  
    Transparency visibility() { return v; }  
    void set_visibility(Transparency t) { v = t; }  
private:  
    Fl_Color c;  
    char v;  
};
```

Draw red grid

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



Line_style

```
struct Line_style {  
    enum Line_style_type {  
        solid=FL_SOLID,                      // -----  
        dash=FL_DASH,                        // - - -  
        dot=FL_DOT,                          // .....  
        dashdot=FL_DASHDOT,                  // - . - .  
        dashdotdot=FL_DASHDOTDOT,            // -..-..  
    };  
  
    Line_style(Line_style_type ss) :s(ss), w(0) {}  
    Line_style(Line_style_type lst, int ww) :s(lst), w(ww) {}  
    Line_style(int ss) :s(ss), w(0) {}  
  
    int width() const { return w; }  
    int style() const { return s; }  
  
private:  
    int s;  
    int w;  
};
```

Example: colored fat dash grid

```
grid.set_style(Line_style(Line_style::dash,2));
```



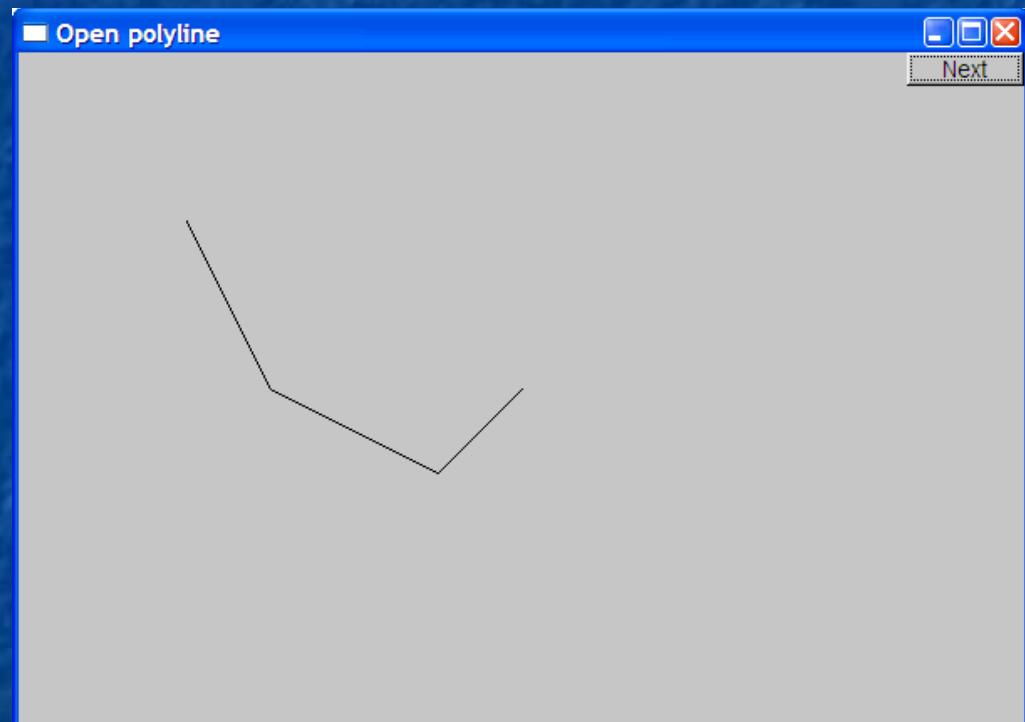
Polylines

```
struct Open_polyline : Shape {    // open sequence of lines
    void add(Point p) { Shape::add(p); }
};

struct Closed_polyline : Open_polyline { // closed sequence of lines
    void draw_lines() const
    {
        Open_polyline::draw_lines(); // draw lines (except the closing one)
        // draw the closing line:
        fl_line( point(number_of_points()-1).x,
                  point(number_of_points()-1).y,
                  point(0).x,
                  point(0).y
        );
    }
    void add(Point p) { Shape::add(p); }    // not needed (why?)
};
```

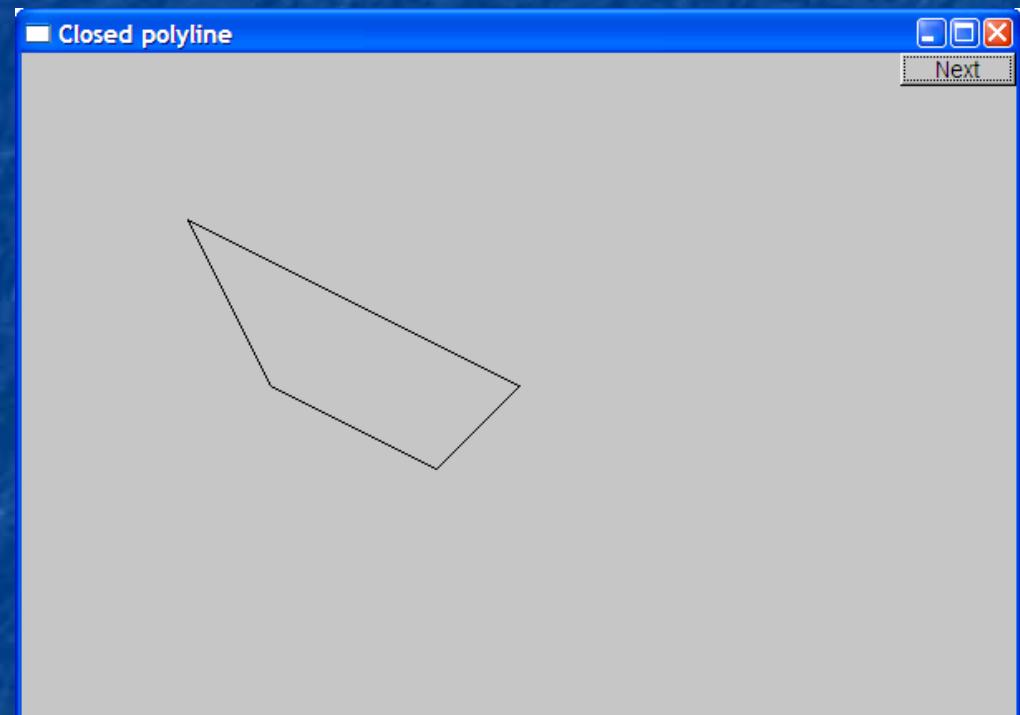
Open_polyline

```
Open_polyline opl;  
opl.add(Point(100,100));  
opl.add(Point(150,200));  
opl.add(Point(250,250));  
opl.add(Point(300,200));
```



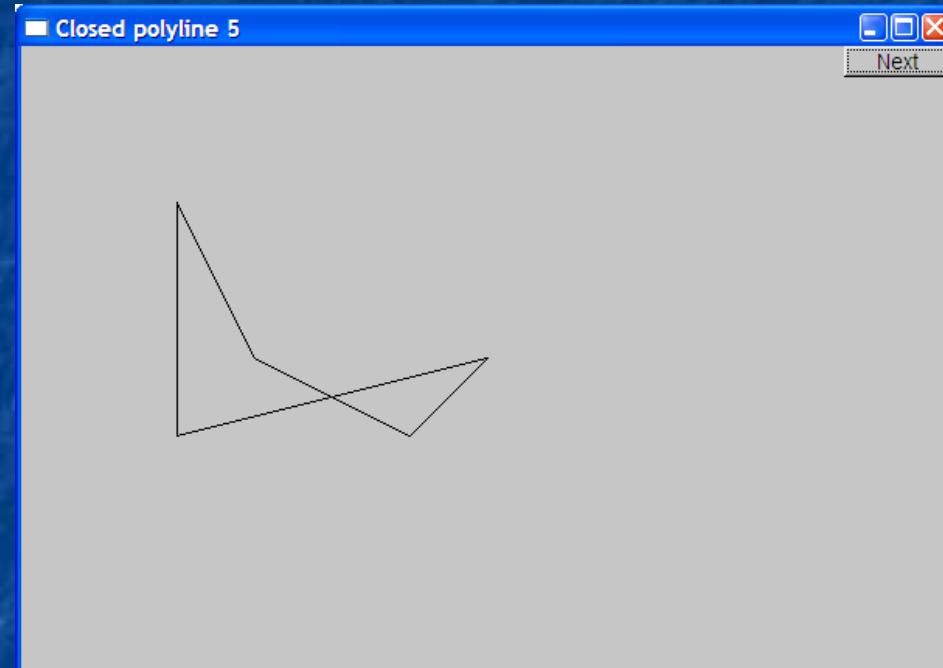
Closed_polyline

```
Closed_polyline cpl;  
cpl.add(Point(100,100));  
cpl.add(Point(150,200));  
cpl.add(Point(250,250));  
cpl.add(Point(300,200));
```



Closed_polyline

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



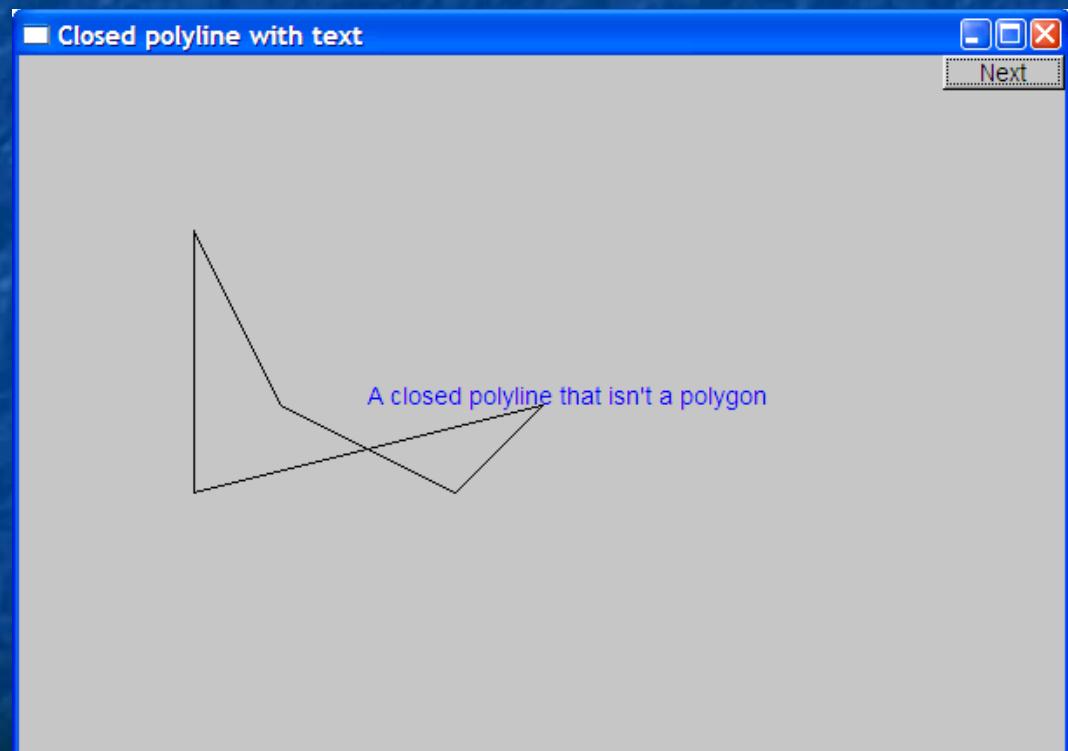
- A **Closed_polyline** is not a polygon
 - some closed_polylines look like polygons
 - A **Polygon** is a **Closed_polyline** where no lines cross
 - A **Polygon** has a stronger invariant than a **Closed_polyline**

Text

```
struct Text : Shape {  
    Text(Point x, const string& s) // x is the bottom left of the first letter  
        : lab(s),  
        fnt(f1_font()), // default character font  
        fnt_sz(f1_size()) // default character size  
    { add(x); } // store x in the Shape part of the Text object  
  
    void draw_lines() const;  
  
    // ... the usual "getter and setter" member functions ...  
  
private:  
    string lab; // label  
    Font fnt; // character font of label  
    int fnt_sz; // size of characters in pixels  
};
```

Add text

```
Text t(Point(200,200), "A closed polyline that isn't a polygon");  
t.set_color(Color::blue);
```

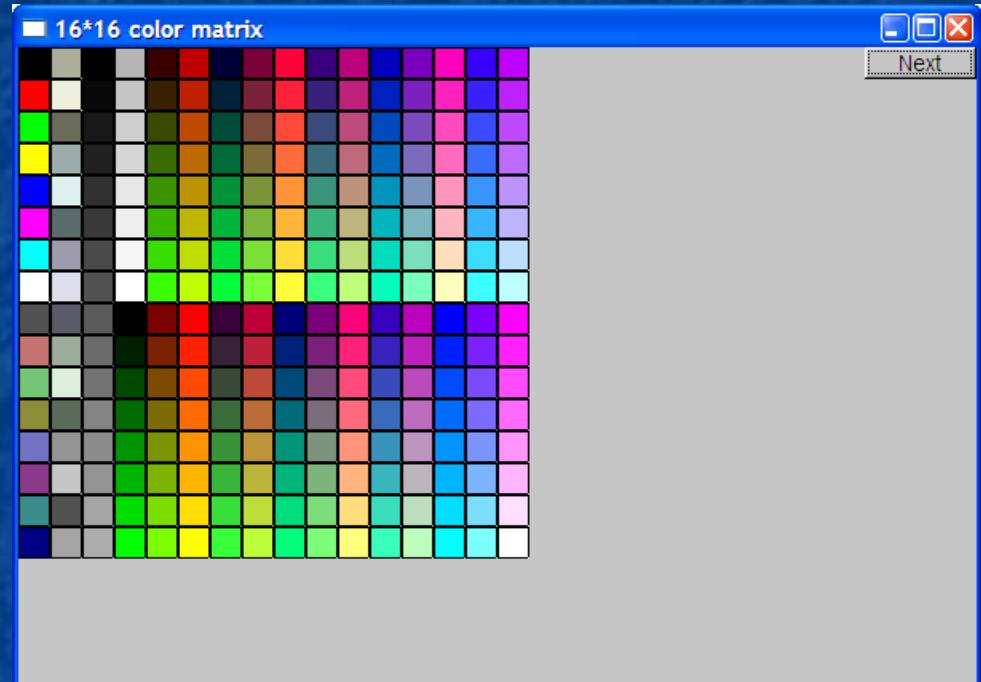


Implementation: Text

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

// fl_draw() is a basic text drawing function from FLTK

Color matrix



Let's draw a color matrix

- To see some of the colors we have to work with
- To see how messy two-dimensional addressing can be
 - See Chapter 24 for real matrices
- To see how to avoid inventing names for hundreds of objects

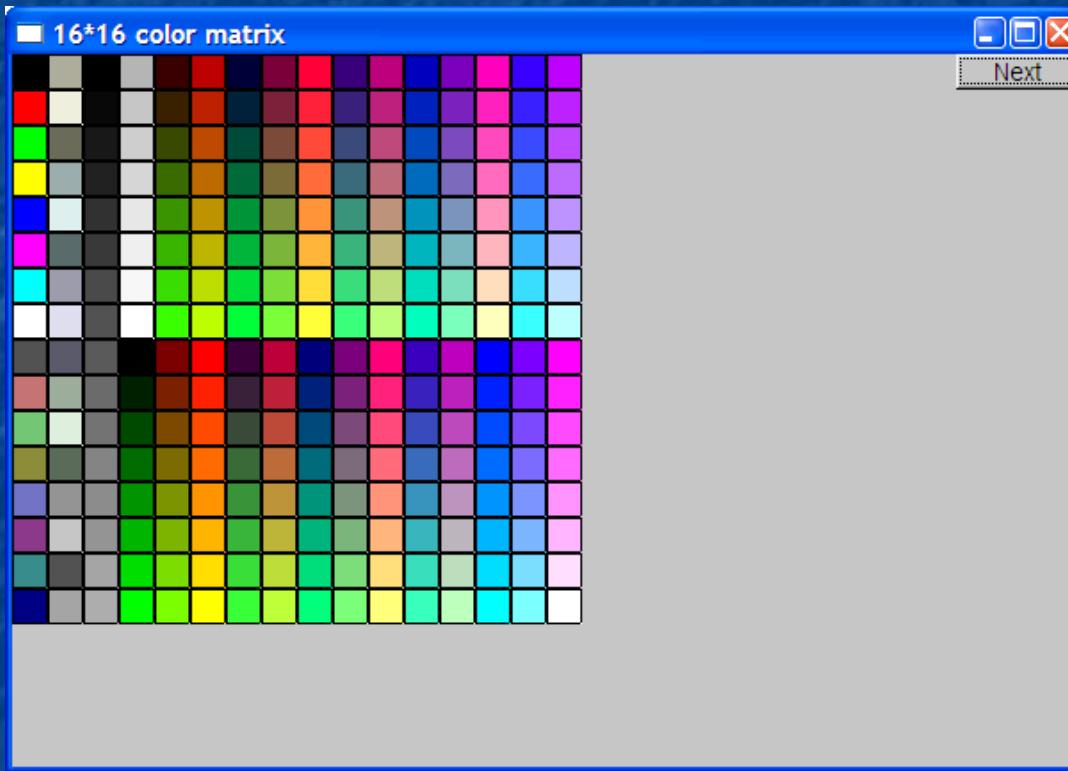
Color Matrix (16*16)

```
Simple_window win20(pt,600,400,"16*16 color matrix");
```

```
Vector_ref<Rectangle> vr; // use like vector  
// but imagine that it holds references to objects  
for (int i = 0; i<16; ++i) { // i is the horizontal coordinate  
    for (int j = 0; j<16; ++j) { // j is the vertical coordinate  
        vr.push_back(new Rectangle(Point(i*20,j*20),20,20));  
        vr[vr.size()-1].set_fill_color(i*16+j);  
        win20.attach(vr[vr.size()-1]);  
    }  
}
```

```
// new makes an object that you can give to a Vector_ref to hold  
// Vector_ref is built using std::vector, but is not in the standard library
```

Color matrix (16*16)



More examples and graphics classes in the book (chapter 13)

Next lecture

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