Graphics Classes

Lecture 17

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

What is Refactoring?

- Changing the structure of code without changing behavior
- Here's the classic example:
  - Refactor

```
if (x)
{
    return true;
}
else
{
    return false;
}
```

  - To

```
return x;
```

Why?

- Refactoring is designed to “clean up” code
- It can become
  - Easier to read
  - Easier to debug
  - Easier to test
  - More efficient
- Refactoring undoes “technical debt”

Technical Debt

- Sometimes you take the “easy approach”
  - You duplicate code
  - You have a complicated base class
  - You hardcode values
  - You use inflexible data structures
  - You use singletons
- You pay for these decisions later

Kinds of Refactoring

- Large refactorings are broken down into micro-refactorings
- Each is guaranteed to preserve semantics
- If you refactor carefully, you won't break your program
  - Run regression tests before refactoring
  - Make one small refactoring change
  - Run regression tests after refactoring

Kinds

- Move Member Function or Data
- Rename Member Function or Data
- Pull-up/push-down
- Extract class/method
- Encapsulate field
- Replacing code with patterns
- Functional Refactoring

Move Member Function or Data

- Why?
  - Each member/method belongs somewhere
  - Once-and-only-once means that we need to find the best place
- Where?
  - Wherever it yields the least coupling (the least amount of other code uses it/depends on it)

Example

```
class Car
{
    static int NUMBER_OF_WHEELS = 4;
};

class CarFactory
{
    static int NUMBER_OF_WHEELS = 4;
};
```

- Where does `NUMBER_OF_WHEELS` belong?

It Depends …

- If `Car` often references the constant, it should have it
- If other classes mainly use the constant, choose the owner who will result in the smallest number of coupled classes
- Basically: if other classes depend on `Car` and want to use the constant, it should belong to `Car`, and vice versa

Rename Member Function or Data

- Choose the name that most closely and specifically reflects what a method does
- Specifically
  - Don't be too vague
  - Don't be too low level
  - This one is subjective
- The most difficult task programmers solve is to come up with new names

Guidelines

- Use the “Rename” function built into your IDE to also catch comments and strings
  - Remember: wrong comments are worse than no comments
- Make names descriptive and accurate:
  - `compute_height( ... ) { height = ... }` is better than
  - `get_height( ... ) { height = ... }`
  - Because “get” implies that you’re returning something instead of setting something
- Don’t be afraid to use longer names
  - Intellisense in modern IDEs helps avoiding to type those over and over again

Pull-up/push-down

- Move a member function or data item from a sub-class to a base-class
- The purpose is to avoid repetition
- But…
  - Consider when you should refactor into a pattern
  - You may find that you don't need inheritance after all (good!)

Pull-up/push-down applies to Hierarchies

- Push state (data members) as far down (away from the base class) as you can
- Why?
  - State changes will require the least amount propagation
  - You will end up with the least amount of coupling

Graphics Classes

Abstract

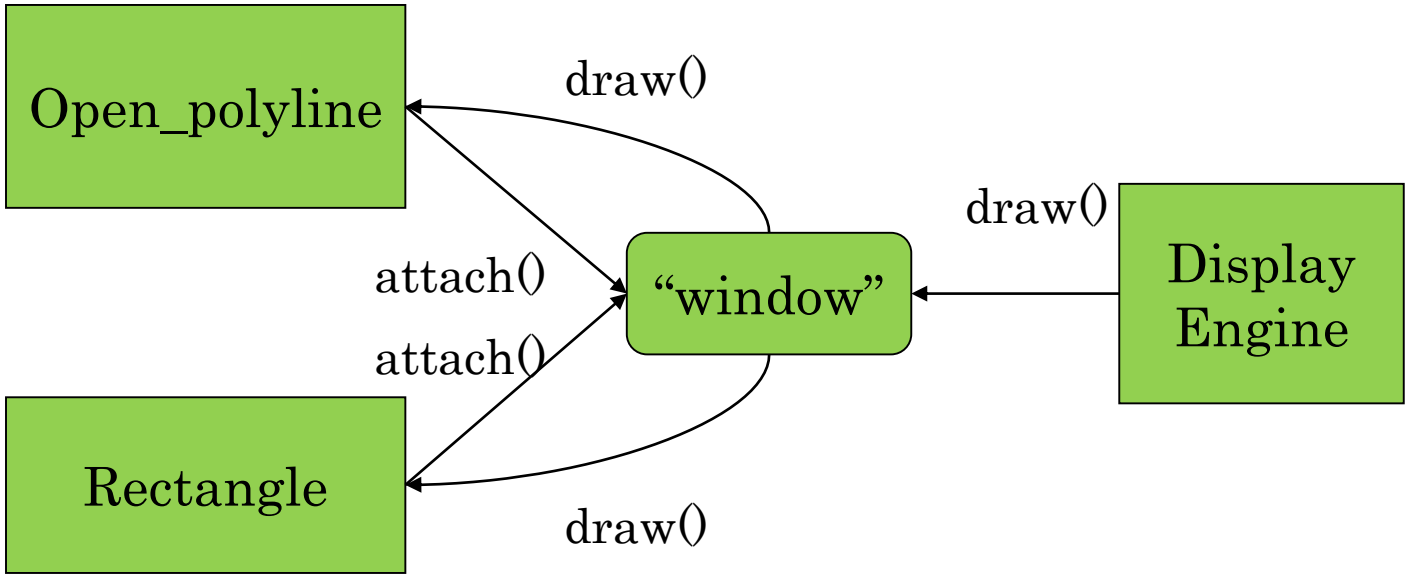
- This lecture presents a display model (the output part of a GUI), giving examples of use and fundamental notions such as screen coordinates, lines, and color. Some examples of shapes are Lines, Polygons, Axis, and Text.

Overview

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

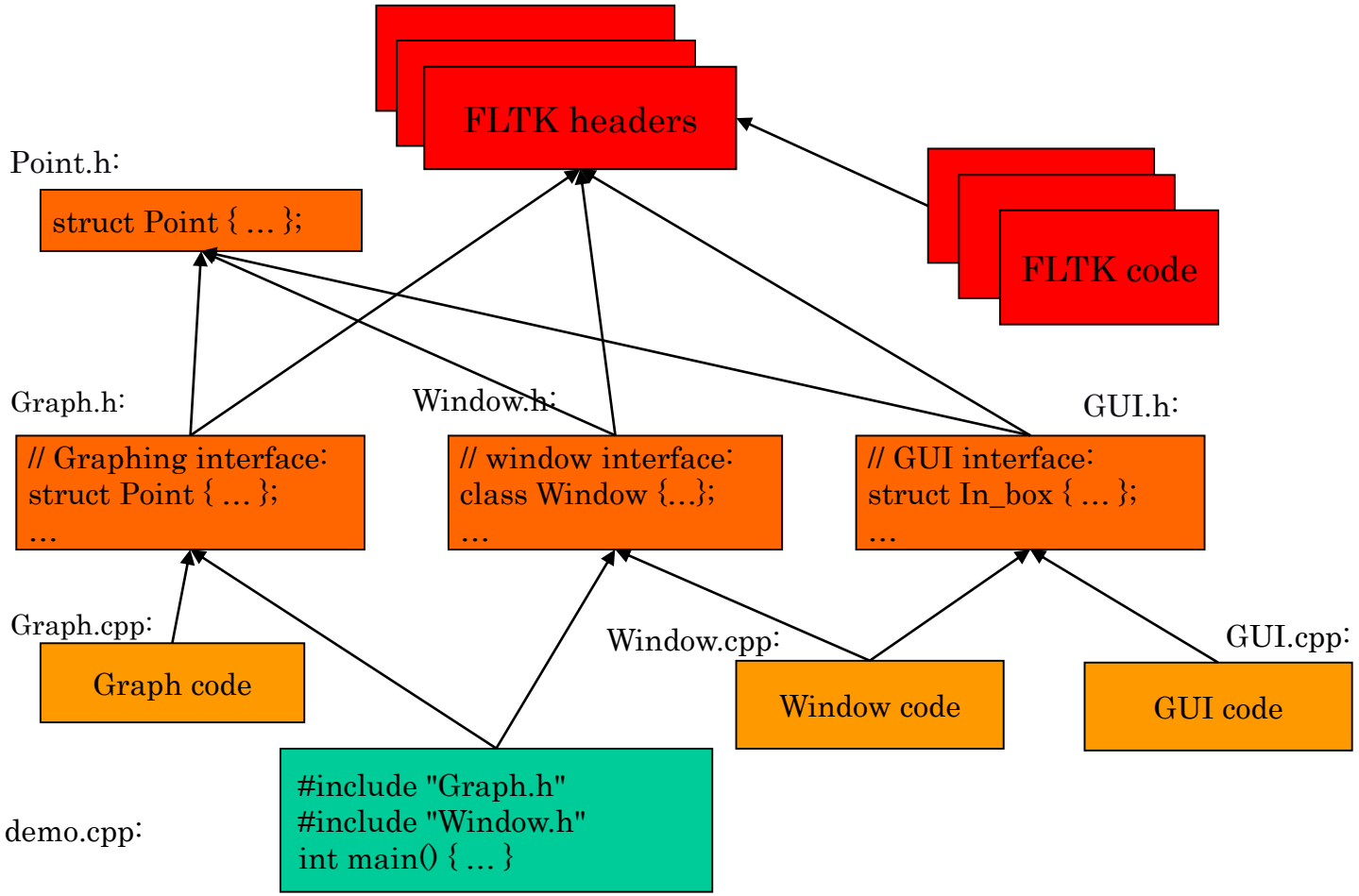
Graphing Classes

The 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: .h, .hpp
  - 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)
  - Point – a coordinate point
  - Line – a line as you see it in a window on the screen
  - Shape – what's common to shapes
    - (imperfectly explained for now; all details in next lecture)
  - 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;
    };

    bool operator==(Point rhs, Point lhs)
    {
        return rhs.x == lhs.x && rhs.y == lhs.y;
    }
    bool operator!=(Point rhs, Point lhs)
    {
        return !(rhs == lhs);
    }
}    // namespace Graph_lib
```

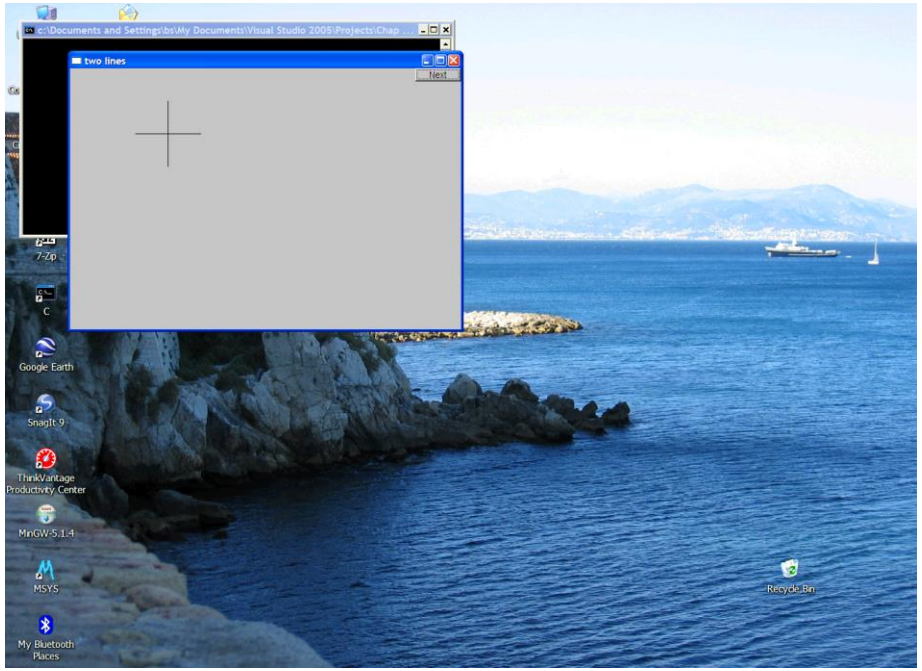
Line

```
struct Shape
{
    // hold lines represented as sequence 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
horizontal.set_color(Color::black);

Line vertical(Point(150, 50), Point(150, 150));            // make a vertical line
vertical.set_color(Color::black);

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

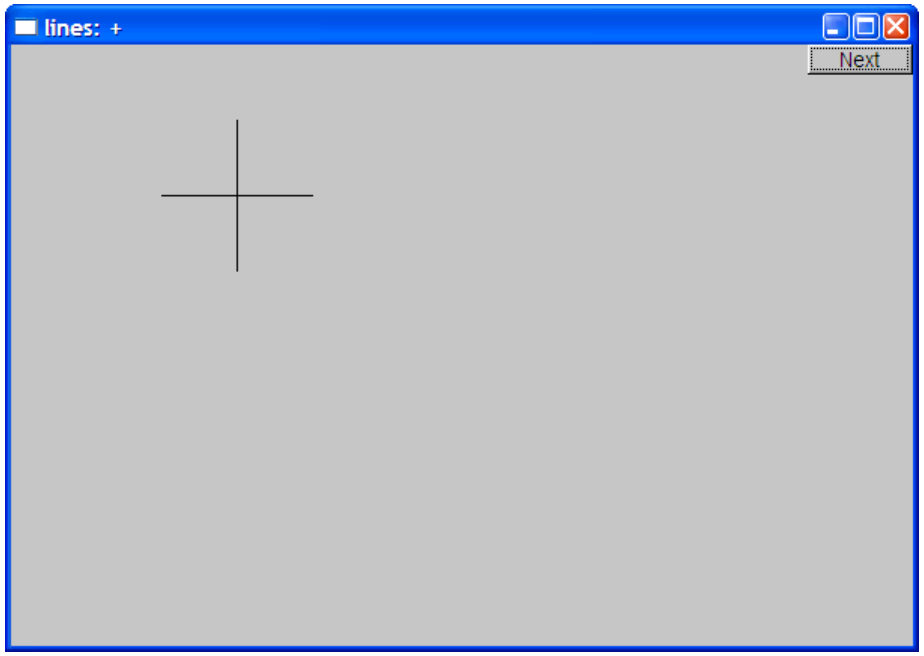
```
// a Lines object is a set of lines
// We use Lines when we want to manipulate
// all the lines as one shape, e.g. move them all
struct Lines : Shape
{
    void add(Point p1, Point p2);    // add line from p1 to p2
    void draw_lines() const;         // to be called by Window to draw Lines
};
```

- Terminology:
  - Lines is “derived from” Shape
  - Lines “inherit 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 later

Lines Example

```
Simple_window win(Point(100, 100), 600, 400, "Canvas");    // make a window

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 x to Window win
win.wait_for_button();      // Draw!
```

Lines example



- Looks exactly like the two Lines example

Implementation: Lines

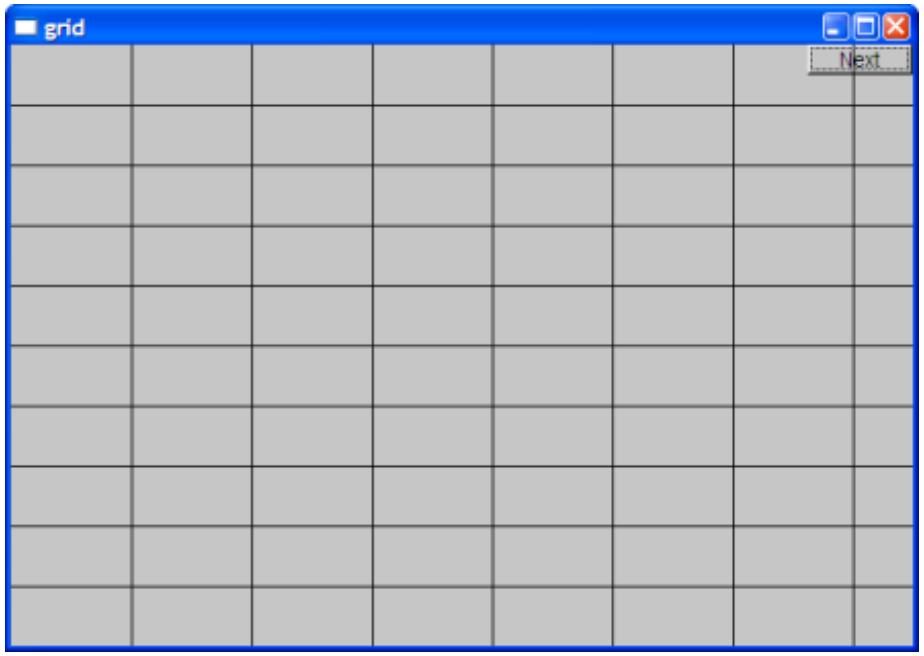
```
void Lines::add(Point p1, Point p2)    // use Shape's add()
{
    Shape::add(p1);
    Shape::add(p2);
}

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;
int y_grid = 40;

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

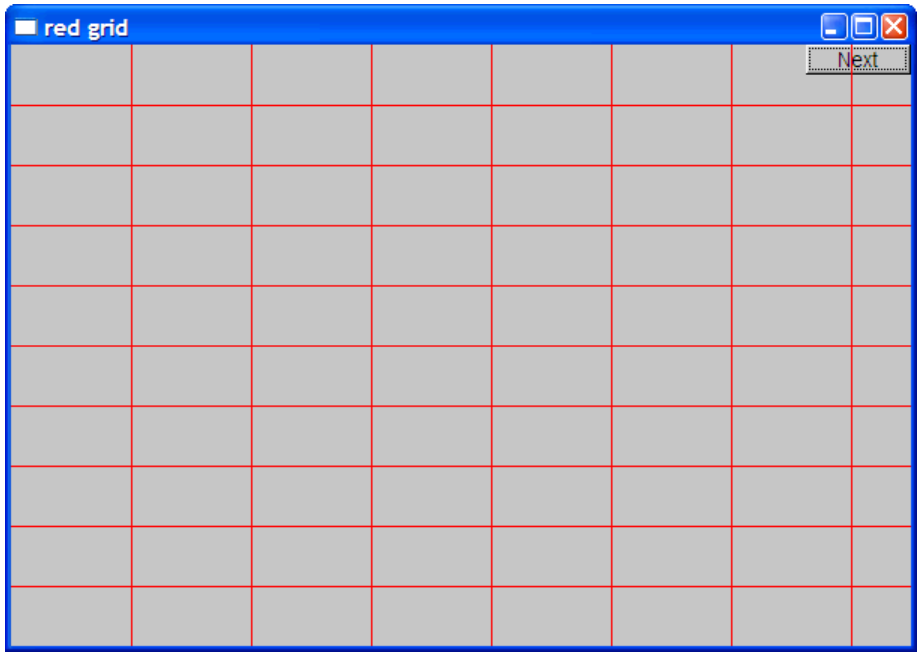


Color

```
struct Color {
    enum Color_type {    // Map FLTK colors and scope them
        red = FL_RED, blue = FL_BLUE, /* ... */
    };

    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; }

private:
    Fl_Color c;     // FLTK type representing a color
    bool v;
};
```

Draw red Rrid

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

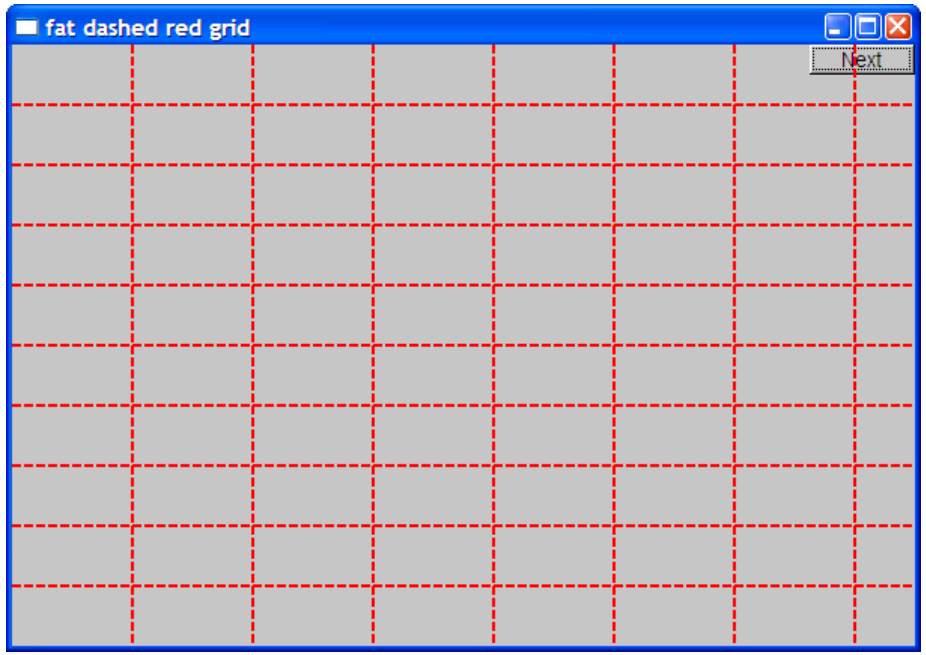


Line_style

```
struct Line_style {
    enum Line_style_type {
        solid = FL_SOLID,           // -------
        dash = FL_DASH,             // - - - -
        dot = FL_DOT,               // .......
    };

    Line_style(Line_style_type lst) : s(lst), 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));
```

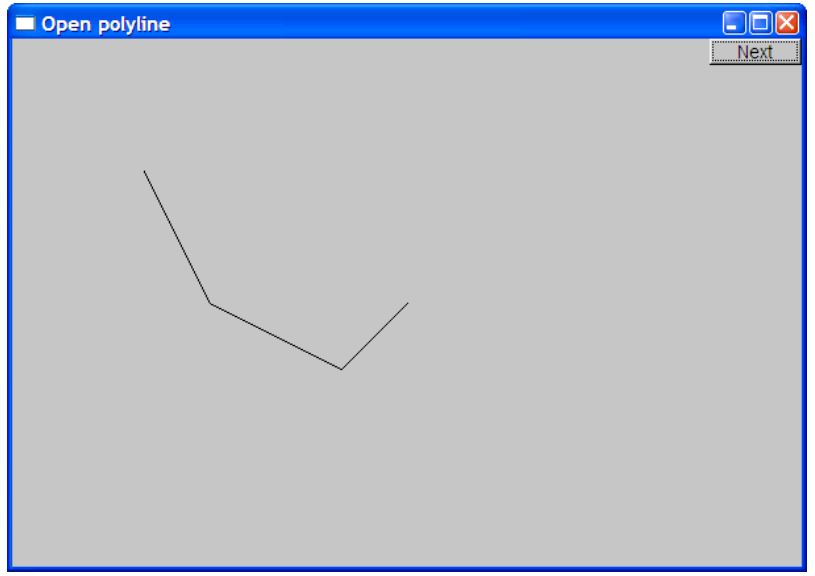


Polylines

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

// closed sequence of lines
struct Closed_polyline : Open_polyline {
    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); }
};
```

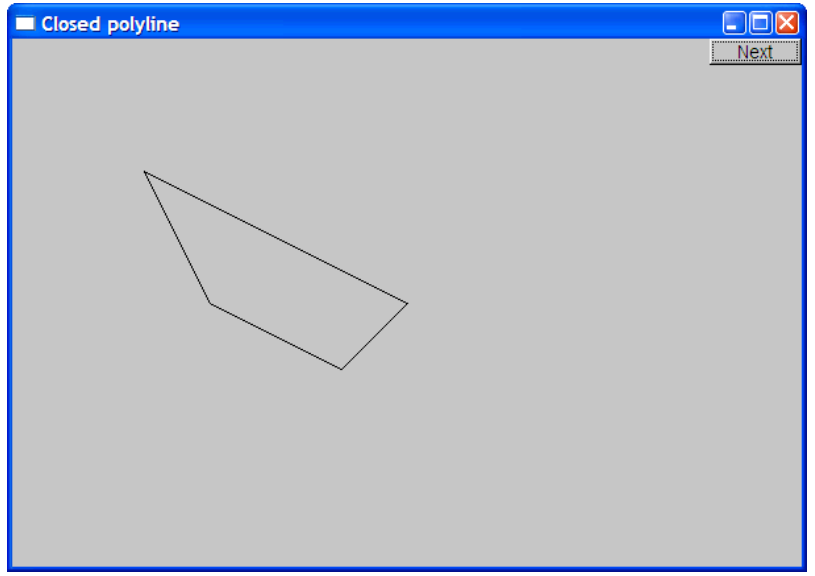
Open_polyline

```
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

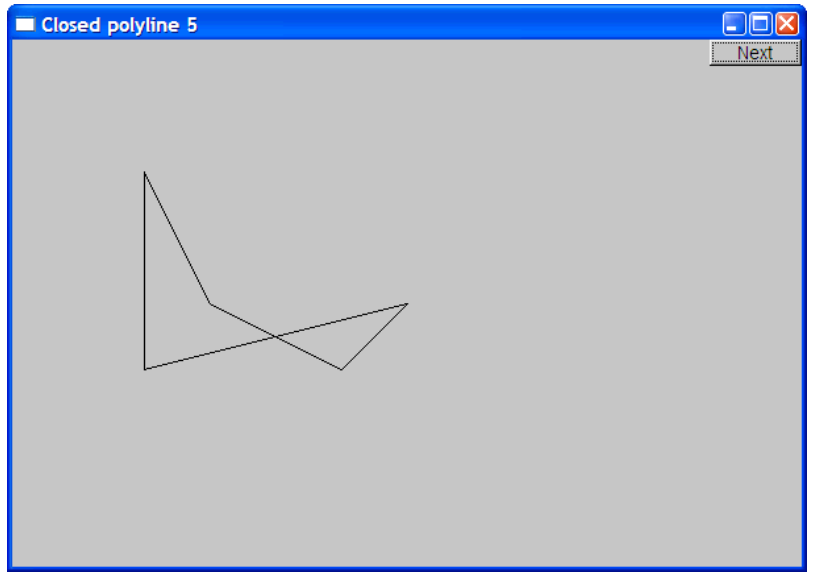
```
Closed_polyline cpl;
cpl.add(Point(100, 100));
cpl.add(Point(150, 200));
cpl.add(Point(250, 250));
cpl.add(Point(300, 200));
```



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

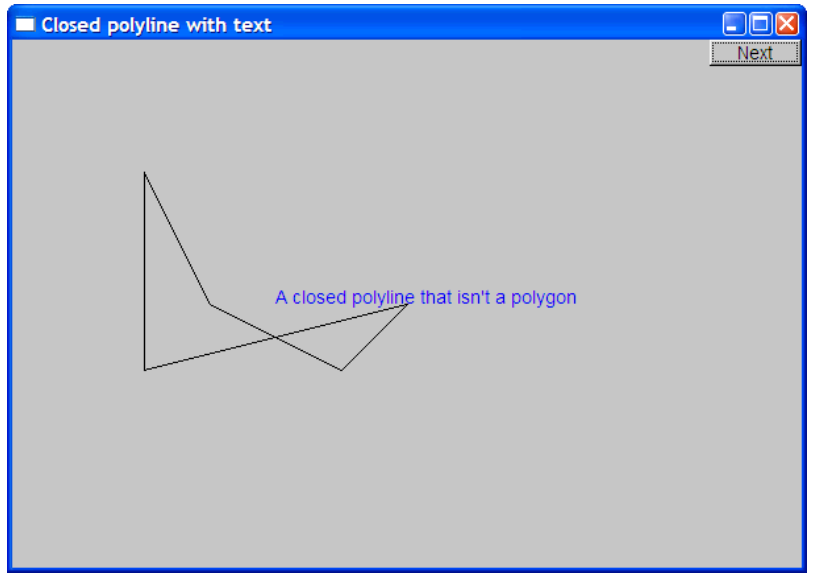


Text

```
struct Text : Shape
{
    // x is the bottom left of the first letter
    Text(Point x, std::string const& s)
      : lab(s)
      , fnt(fl_font())       // default character font
      , fnt_sz(fl_size())    // default character size
    {
        Shape::add(x);    // store x in the Shape part of the Text
    }

    void draw_lines() const;

private:
    std::string lab;    // label
    Font fnt;           // character font of label
    int fnt_sz;         // size of characters
};
```

Add Text

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

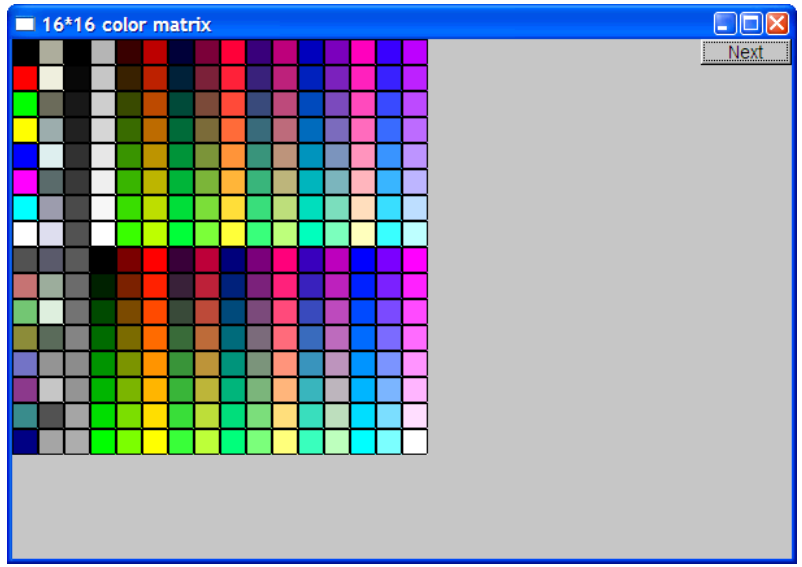


Implementation: Text

```
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
  - To see how to avoid inventing names for hundreds of objects



Color Matrix (16*16)

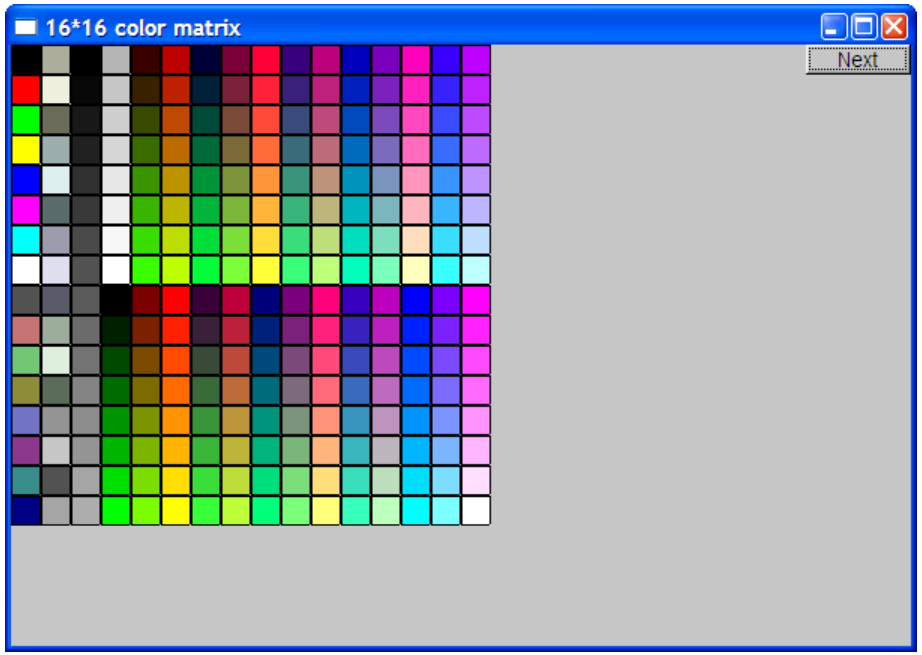
```
Simple_window win(pt, 600, 400, "16*16 color matrix");

// normal vector
std::vector<Rectangle> v;

for (int i = 0; i < 16; ++i)          // i is the horizontal coordinate
{
    for (int j = 0; j < 16; ++j)    // j is the vertical coordinate
    {
        v.push_back(Rectangle(Point(i * 20, j * 20), 20, 20);
        v.back().set_fill_color(i * 16 + j);
        win.attach(v.back());
    }
}

// make_unique makes an object that you can give to a unique_ptr to hold
// unique_ptr behaves like a pointer, but manages memory for you
```

Color Matrix (16*16)

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

// normal vector, but imagine that it holds references to objects
std::vector<std::unique_ptr<Shape>> v;

for (int i = 0; i < 16; ++i)          // i is the horizontal coordinate
{
    for (int j = 0; j < 16; ++j)    // j is the vertical coordinate
    {
        v.push_back(std::make_unique<Rectangle>(Point(i * 20, j * 20), 20, 20));
        v.back()->set_fill_color(i * 16 + j);
        win.attach(*v.back());
    }
}

// make_unique makes an object that you can give to a unique_ptr to hold
// unique_ptr behaves like a pointer, but manages memory for you
```

Color matrix (16*16)

- More examples and graphics classes in the book*



*Stroustrup: Programming - Principles and Practice using C++

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP