ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

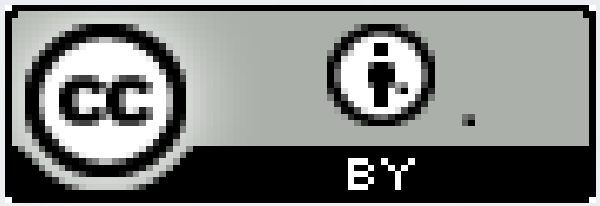
Διάλεξη 12: Κλάσεις Γραφικών

Γ. Παπαγιαννάκης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα (Attribution 3.0– Unported GR)



- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

HY-150 Προγραμματισμός
CS-150 Programming

Lecture 12:
Graphics Classes

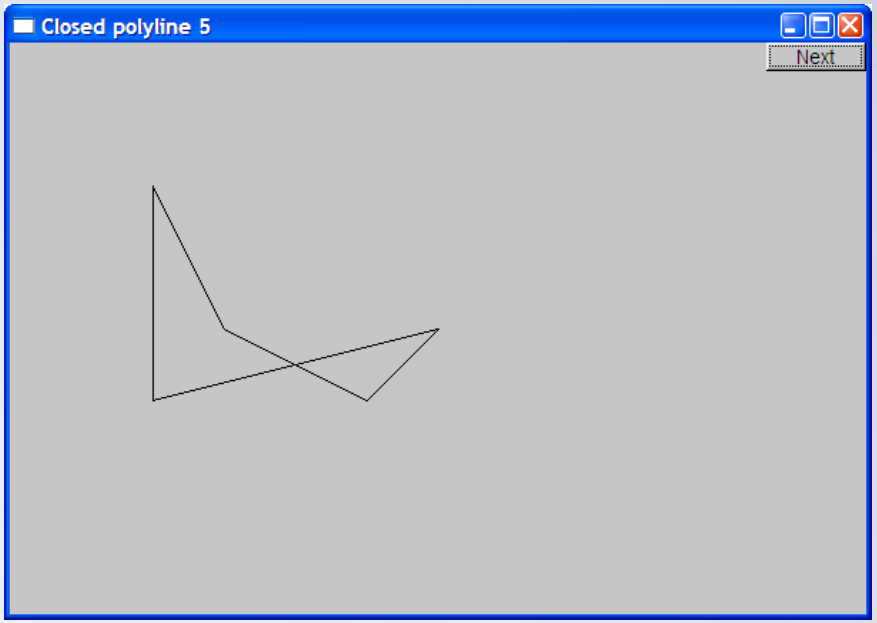
G. Papagiannakis

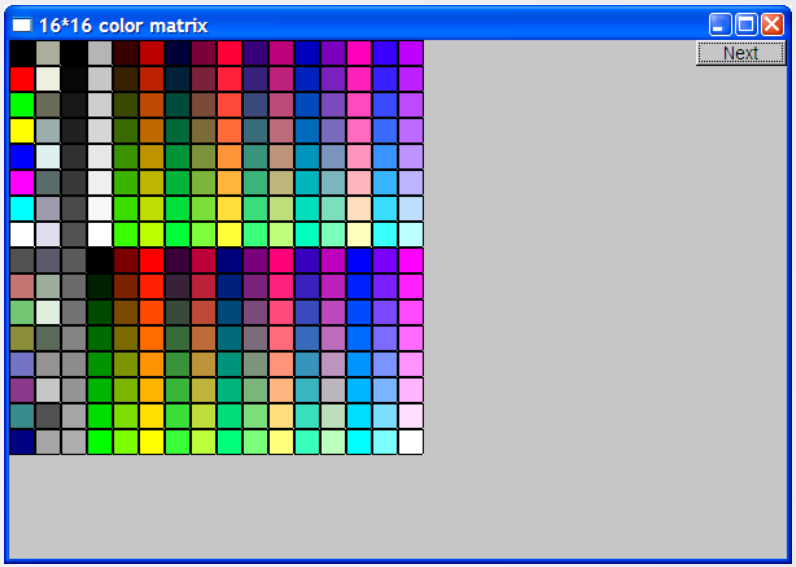
Abstract

- Previous lecture (Chapter 12 in book) demonstrated how to create simple windows and display basic shapes:
  - square, circle, triangle, and ellipse; It showed how to manipulate such shapes: change colors and line style, add text, etc.
- This lecture (Chapter 13 in book) 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.

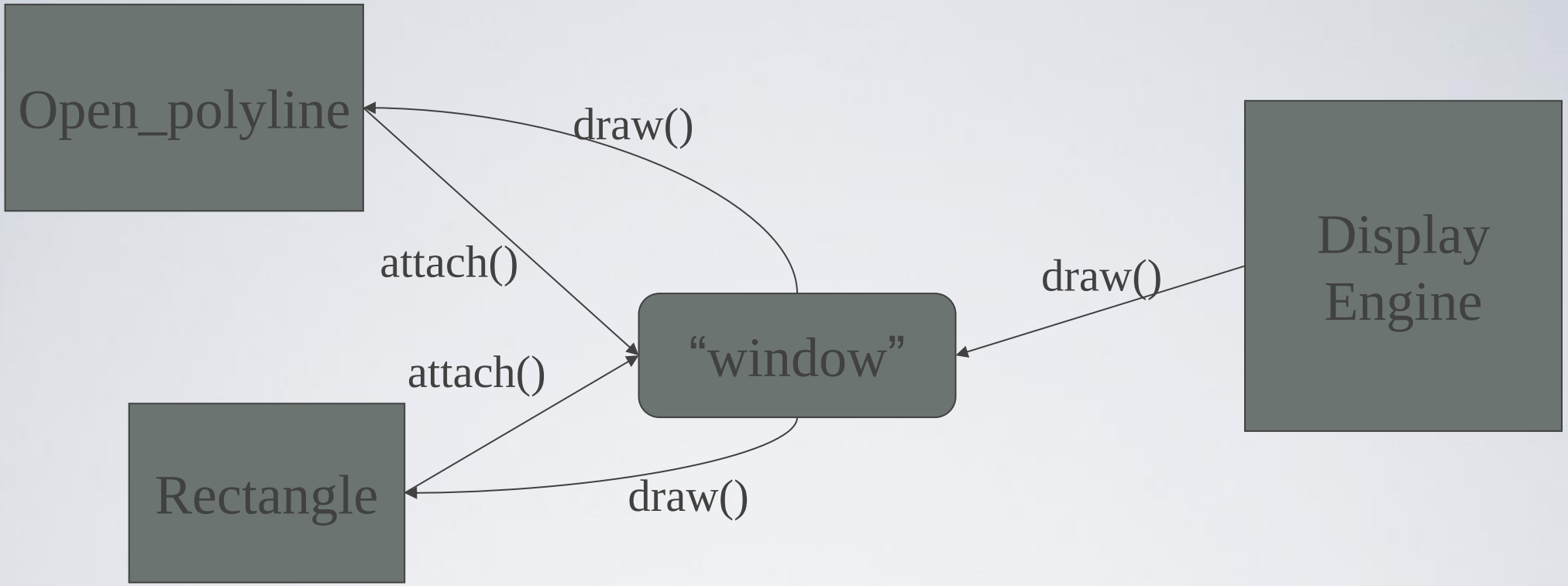
Overview

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



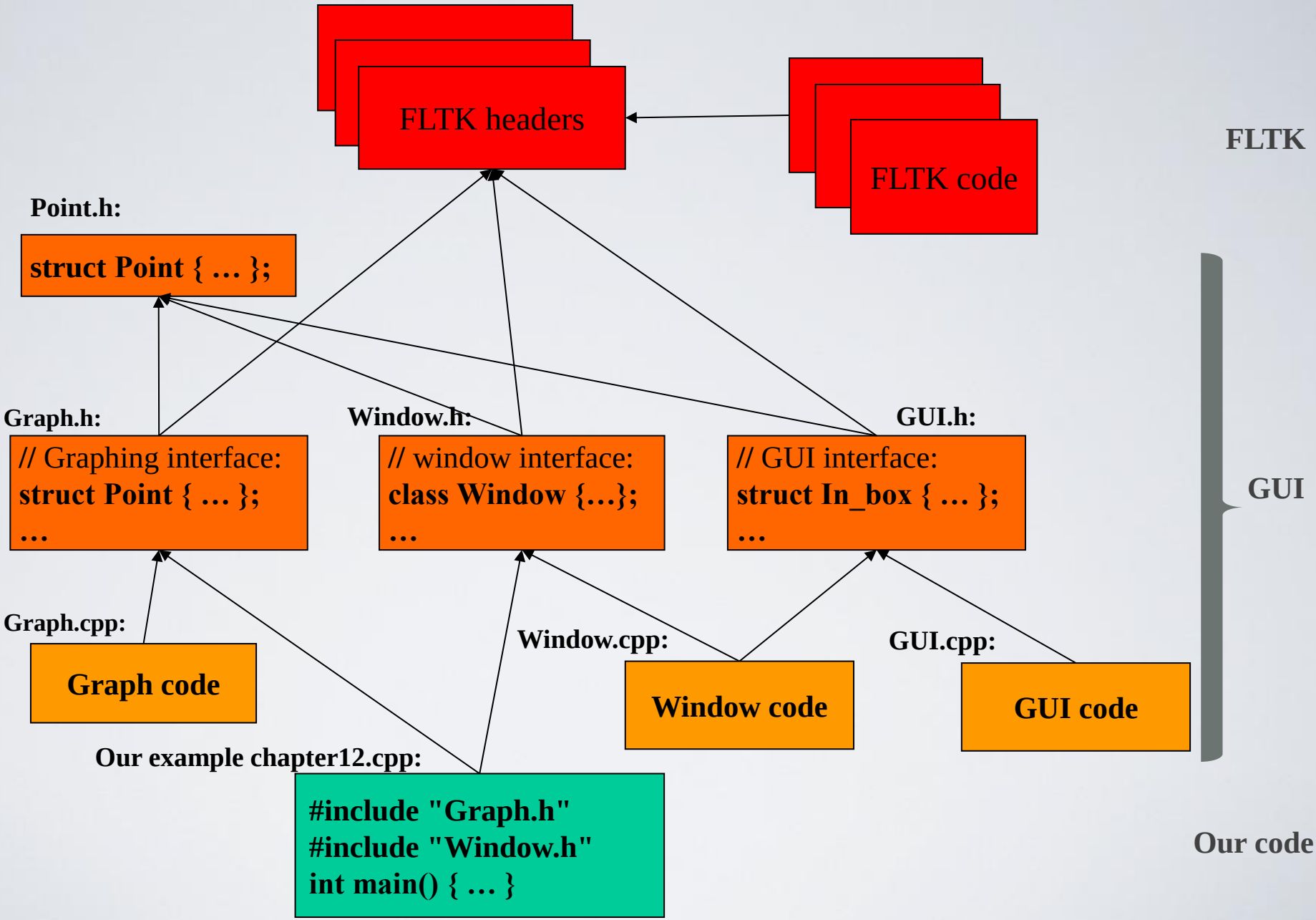


Display model



- 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

Layered 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
  - **#include**s 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) { }
   };                        // Note the ';'
}
```

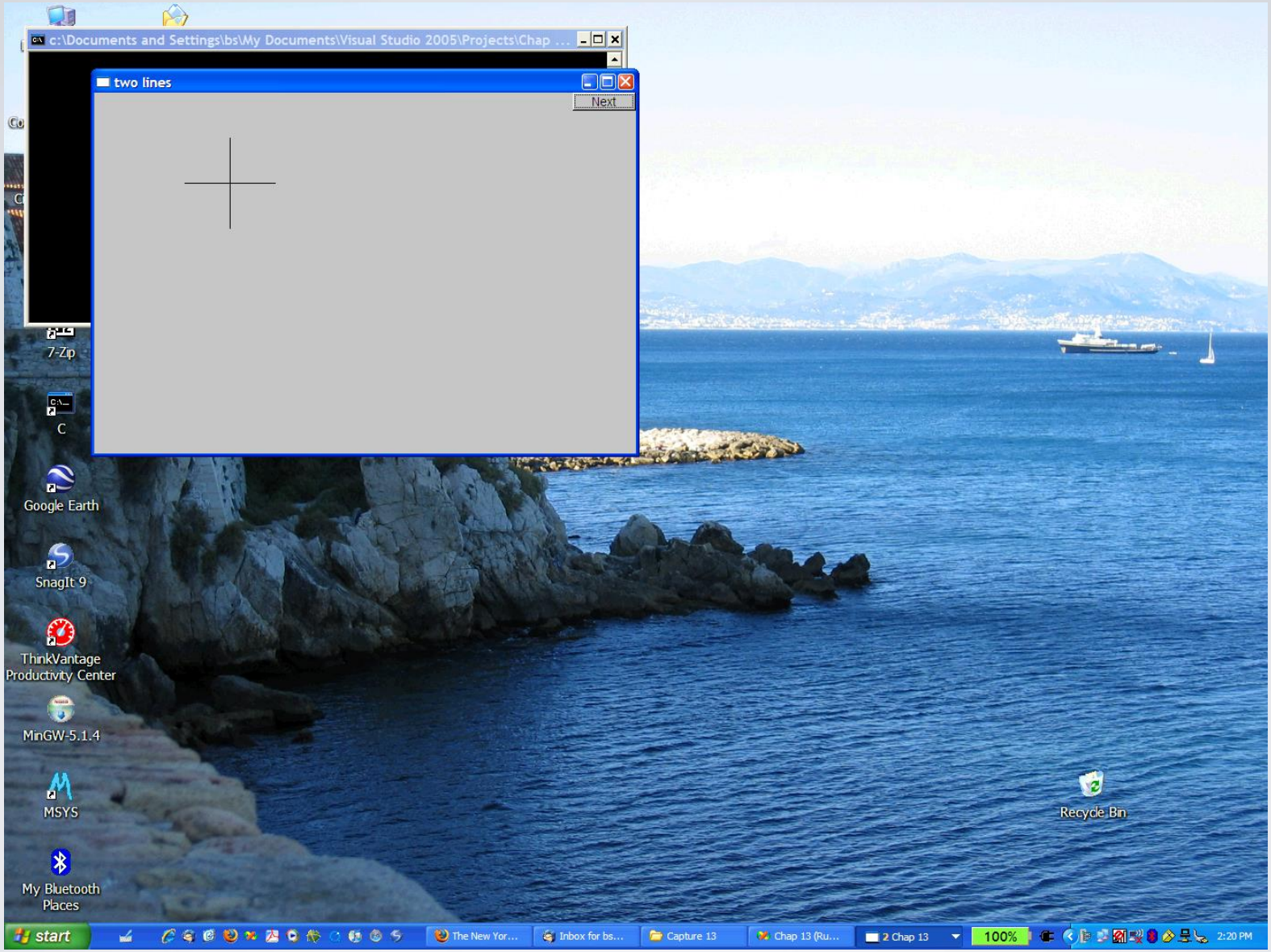
Line

```
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

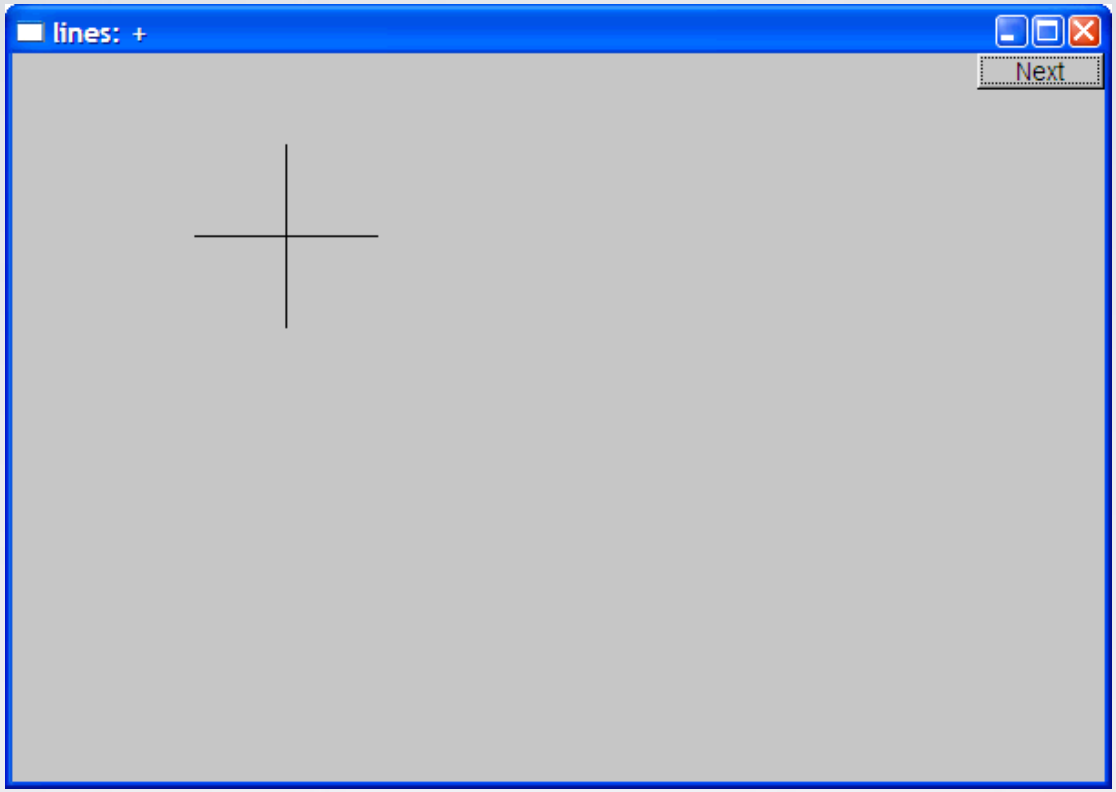
```
struct Lines : Shape {          // a Lines object is a set lines
                                // We use Lines when we want to manipulate
                                // all the lines as one shape, e.g. move them all

    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 *(παράγεται από)*
  - Lines “inherit from” Shape *(κληρονομεί την)*
  - Lines is “a kind of” Shape *(είναι του είδους)*
  - Shape is “the base” of Lines *(είναι η βάση)*
- 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 x to Window win
win.wait_for_button();// Draw!
```

Lines example



- Looks exactly like the two **Line**s 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);
}
```

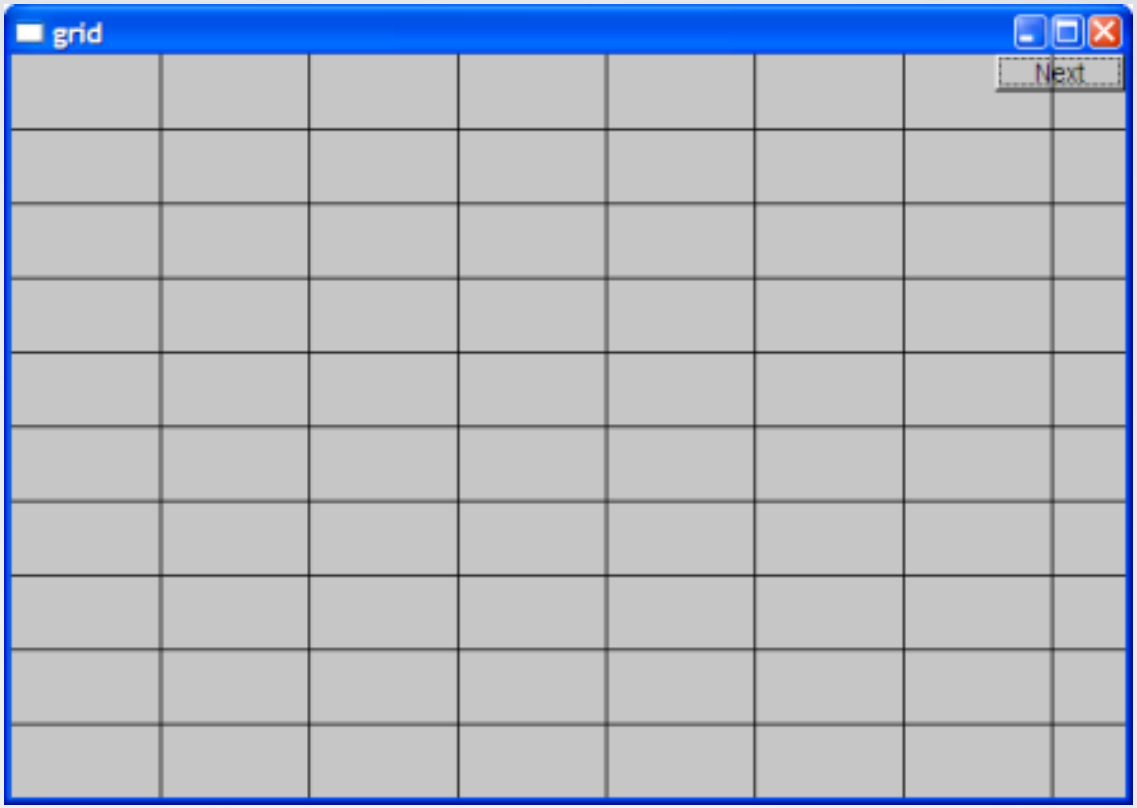
- Note
  - fl_line is a basic line drawing function from FLTK
  - FLTK is used in the *implementation*, not in the *interface* to our classes
  - We could replace FLTK with another graphics library

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)     // veritcal 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

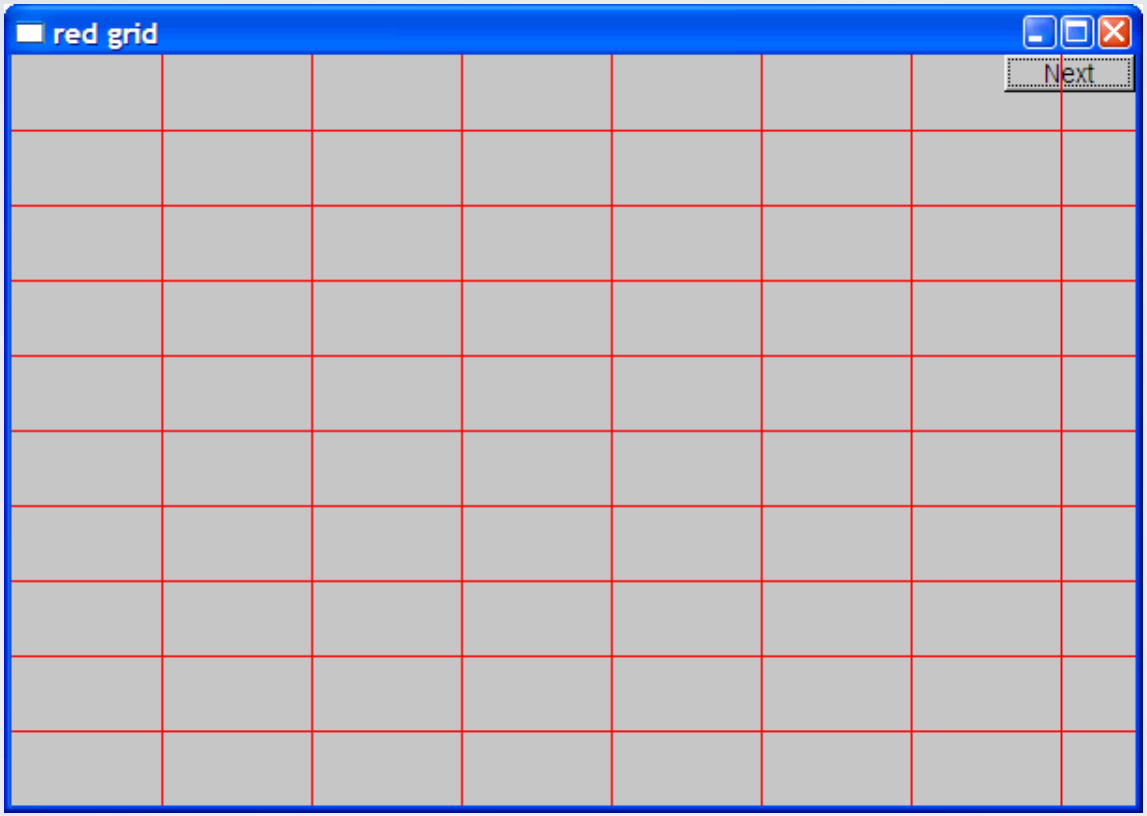


Color

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

   enum Transparency { visible=0, invisible=255 };

   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;
   Transparancy v;         // visibility (transparency not yet implemented)
};
```

Draw red grid

grid.set_color(Color::red);

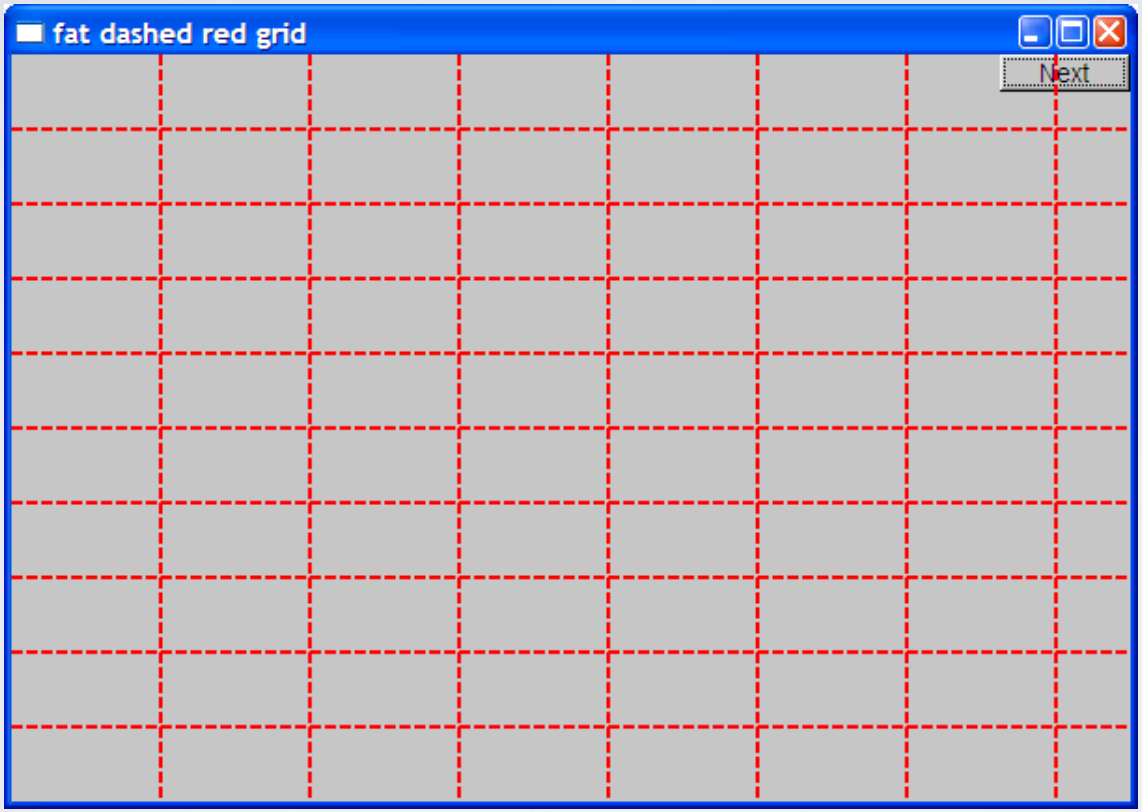


Line_style

```
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));

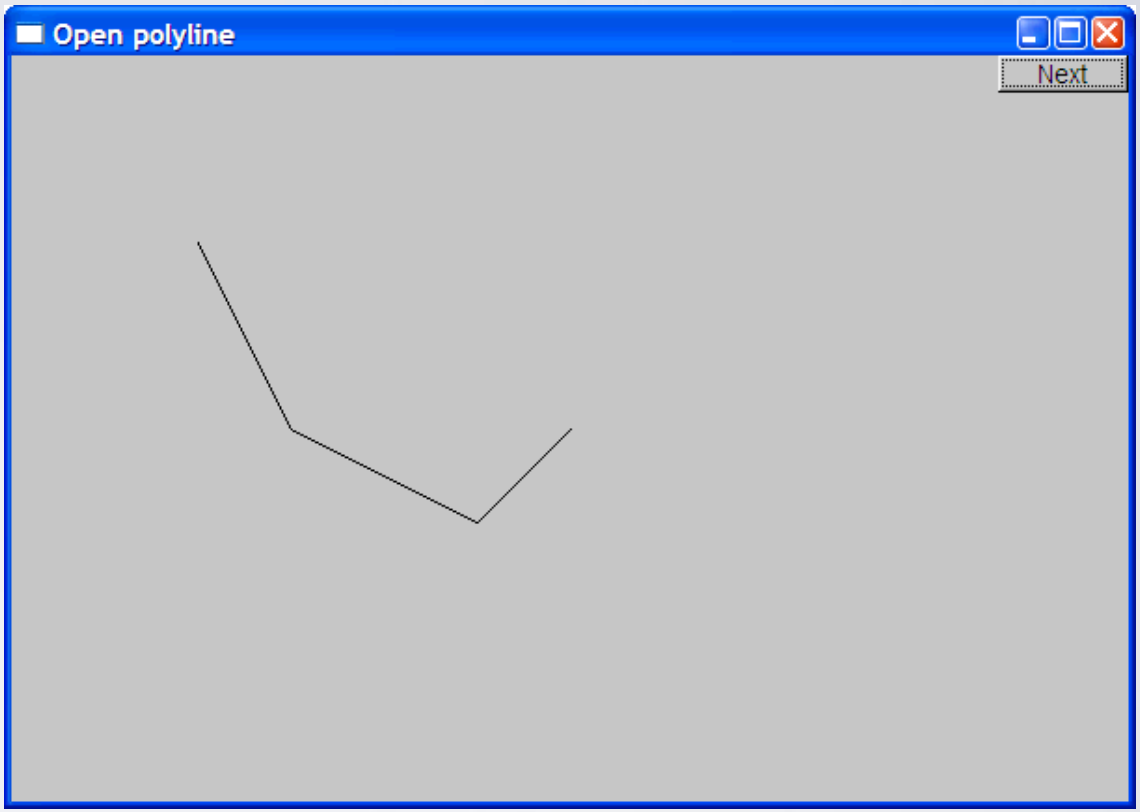


Polylines

```
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); }
};
```

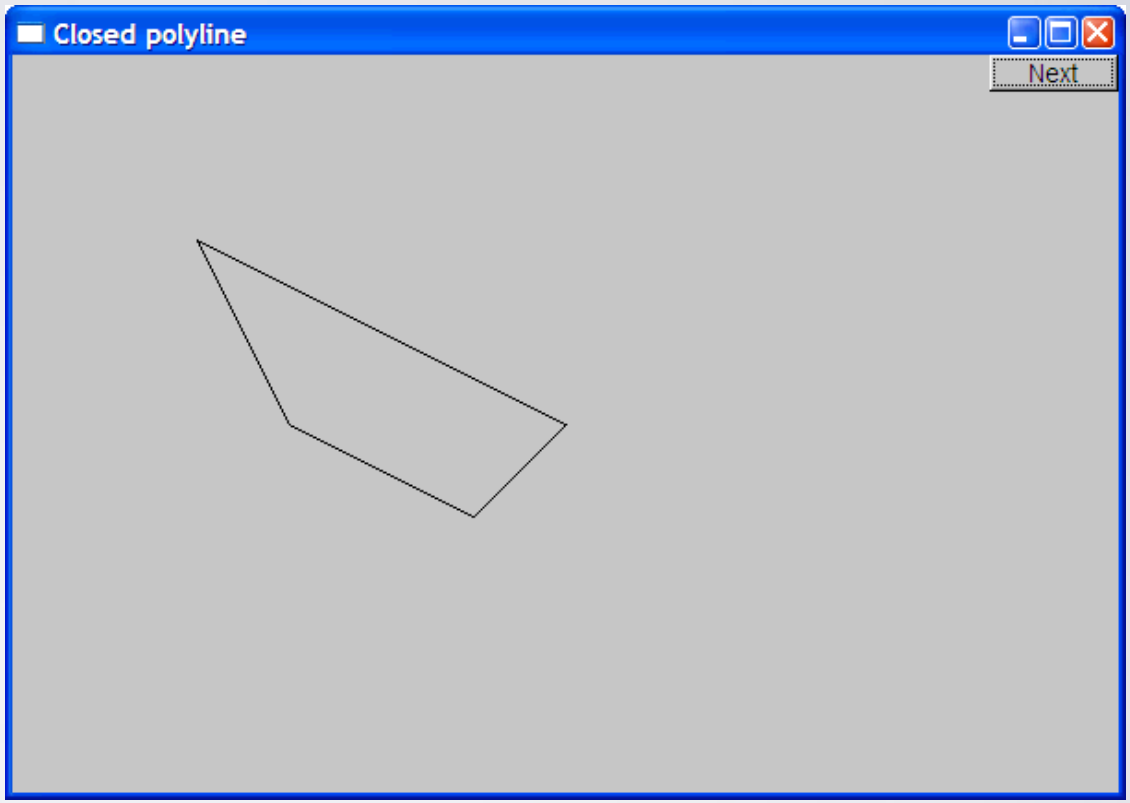
Open_polyline

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



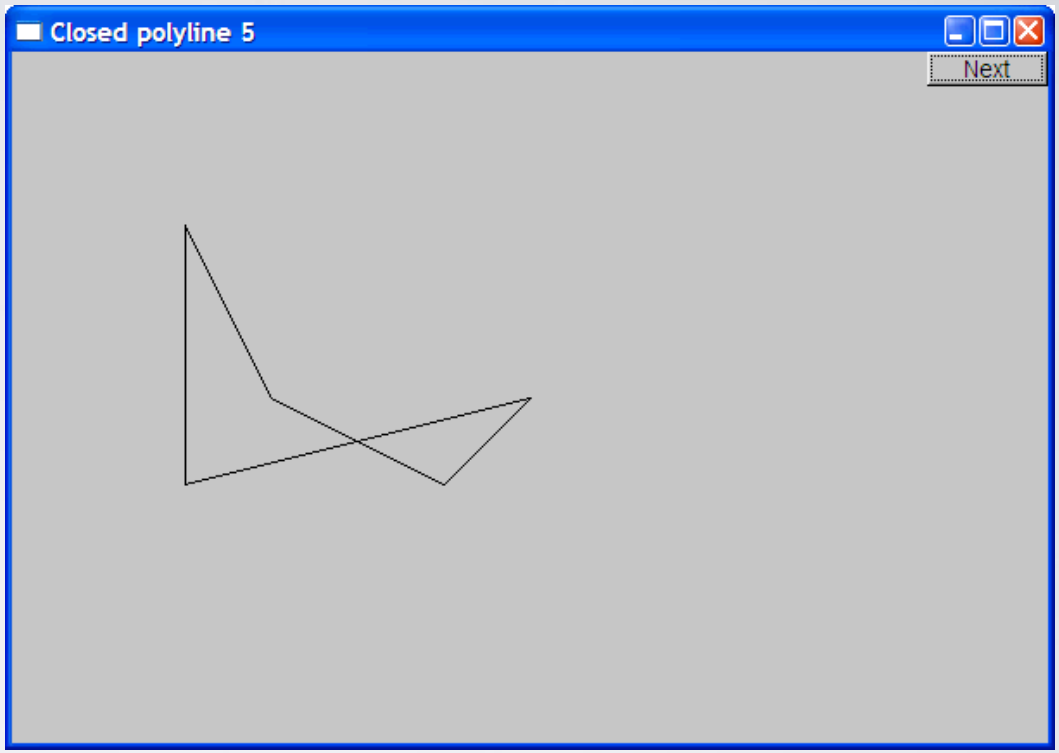
Closed_polyline

```
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**

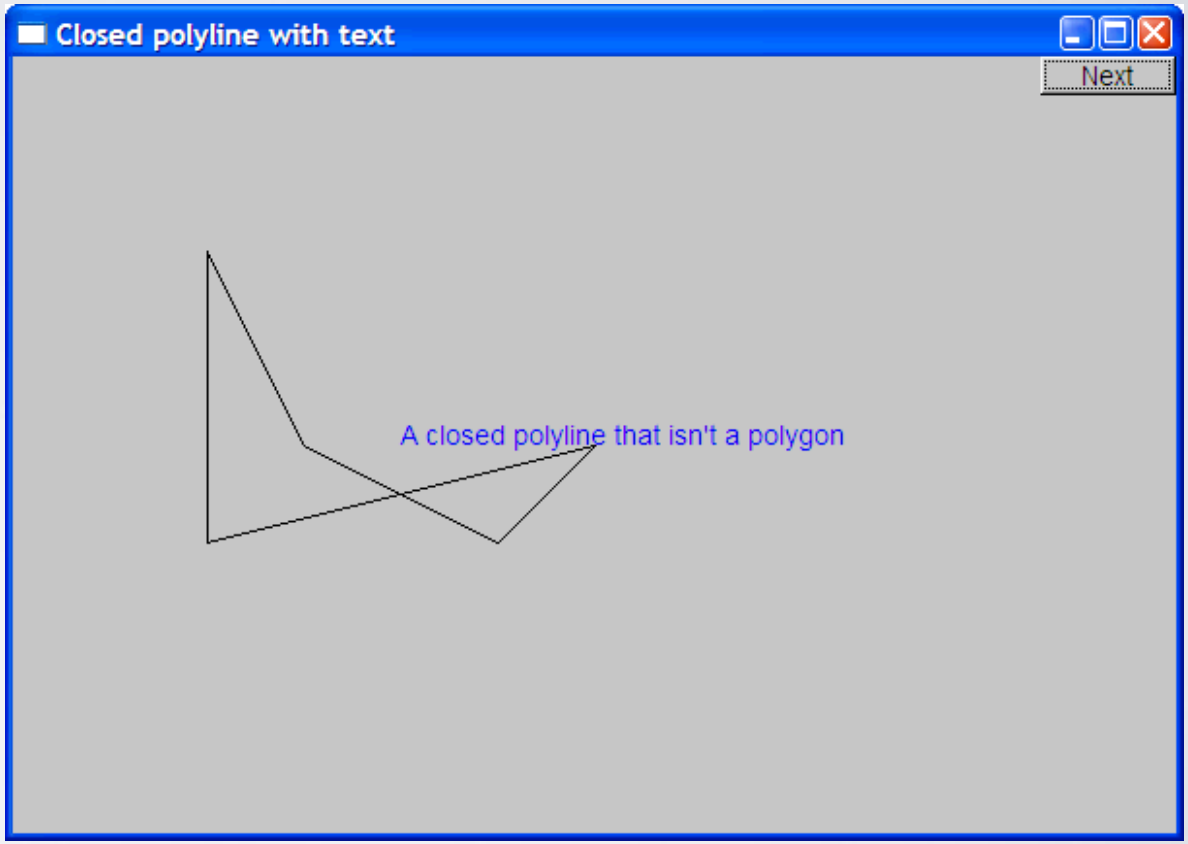
Text

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

    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
};
```

Add text

Text t(Point(200,200), "A closed polyline that isn’t a polygon");

t.set_color(Color::blue);

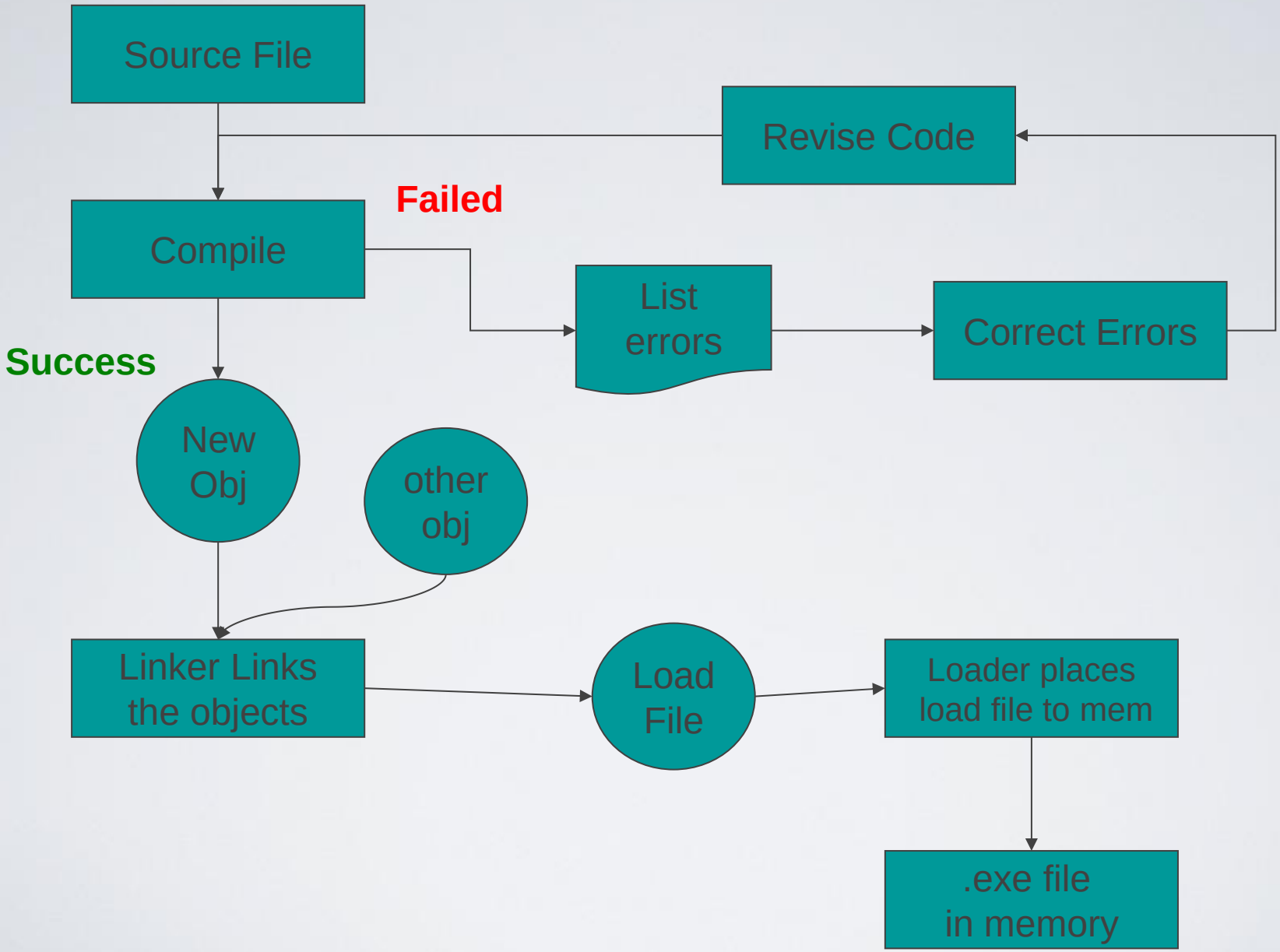


Implementation: Text

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

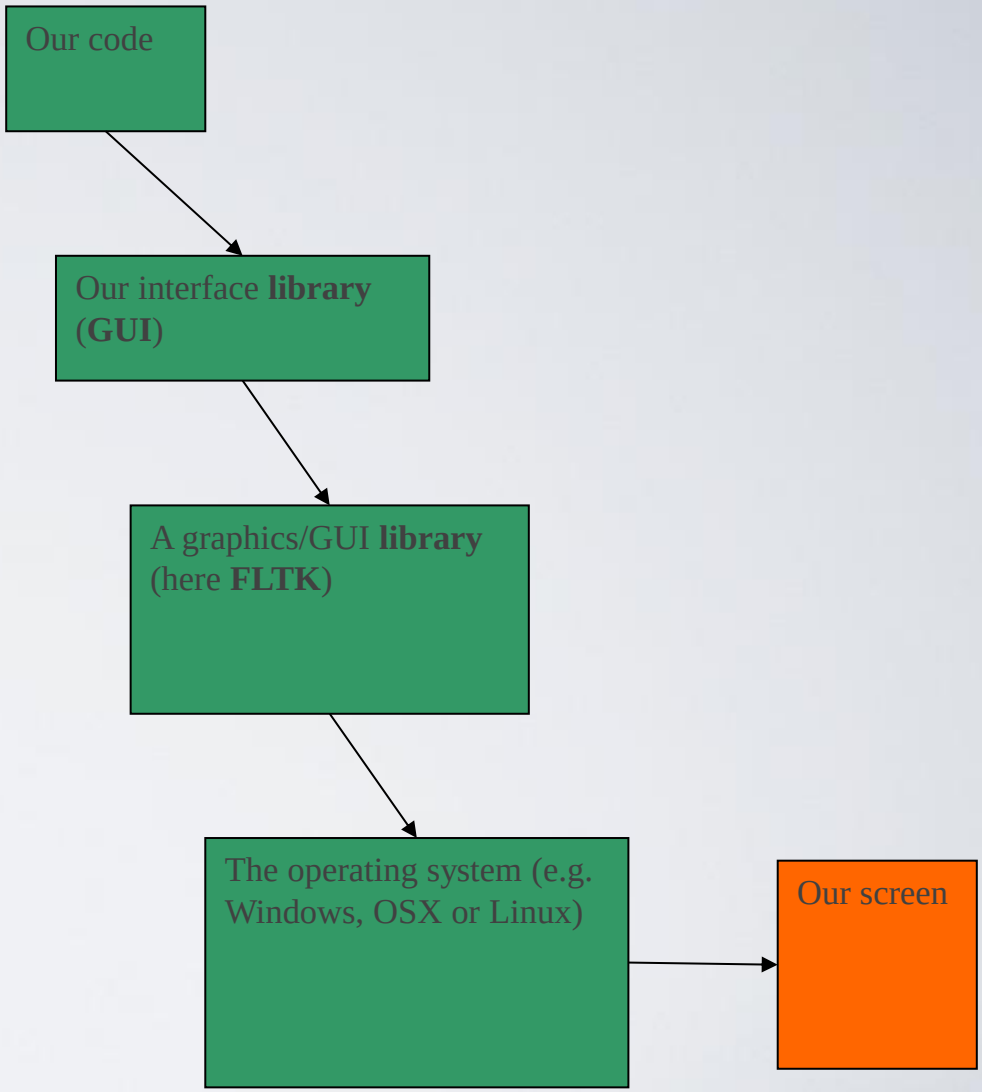
// fl_draw() is a basic text drawing function from FLTK

compiling/linking notes: How .exe file is created?



What are s/w libraries?

- C/C++ libraries are modules of already-compiled code which are accessed through code you will write.
- For example, GUI is a library that provides a lot of the functionality you would need to create FLTK GUIs,
  - allowing you to work with FLTK without having to write directly FLTK code.
- Compiling libraries is much like compiling an executable, only the code gets compiled into a non-executable file instead.



Static & dynamic s/w libraries?

- Libraries come in two different flavors: **static** libraries and **dynamic** libraries.
  - A **static library** is specifically used at *compile time*, which means the libary must be present in the correct location (often but not always somewhere in your project folder) when you want to compile your C/C++ application, since it is *compiled into* the executable, and indeed, the compiler knows it must be there and will not compile without it.
    - Under Windows, static libraries are usually suffixed **.lib**
    - while under the *nixes (includes Android, iOS, MacOS), they are suffixed **.a**
  - A **shared library** is only required to be present for use at *runtime*. Which means you can compile your code without it. Since a shared library is not actually a part of your application's code (although your application may need it to run without failing).
    - When you run your favorite games, you will often see DLL files hanging out in the game's root folder, beside the executable. Those are shared libraries which the executable will need to use when you run it. Under Windows, shared libraries are **.dll**.
    - Under the *nixes, they are **.so**. There are many differences undoubtedly, but the principle is the same.

compiling/linking notes: g++ options I

- build an executable out of one input file (single stage)
  - g++ -Wall -time -o chapter.9.2.exe chapter.9.2.cpp
- build an executable out of one input file (two stages)
  - g++ -Wall -time -O3 -DNDEBUG -o chapter.9.7.5.o -c chapter.9.7.5.cpp
  - g++ -o chapter.9.7.5.exe chapter.9.7.5.o -lstdc++
- build executable out of two files (single stage)
  - g++ -Wall -o Part3 Part3.cpp Rectangle.cpp
- build an executable out of two input files (two stages)
  - g++ -Wall -time -O3 -DNDEBUG -o Part3.o -c Part3.cpp
  - g++ -Wall -time -O3 -DNDEBUG -o Rectangle.o -c Rectangle.cpp
  - g++ -o Part3 Part3.o Rectangle.o -lstdc++
    - //linker options –lstdc++ vary according to different Linux-based systems

g++ FLAGS	
-D	definition
-O3	optimize
-o	object
-c	compile only (not link)
-Wall	all warnings
-time	show time

compiling/linking notes: g++ options II

- How to build a static library (e.g. GUI)
  - g++ -I"" -Wall -time -O3 -DNDEBUG -o GUI.o -c GUI.cpp
  - g++ -I"" -Wall -time -O3 -DNDEBUG -o Graph.o -c Graph.cpp
  - g++ -I"" -Wall -time -O3 -DNDEBUG -o Simple_window.o -c Simple_window.cpp
  - g++ -I"" -Wall -time -O3 -DNDEBUG -o Window.o -c Window.cpp
  - ar rcs libbookgui.a GUI.o Graph.o Simple_window.o Window.o

compiling/linking notes: g++ options III

- How to build an application on top of libraries (e.g. FLTK, GUI)
  - g++ -I"../GUI" -I"" -Wall -time -O3 -DNDEBUG -I/usr/local/include -I/usr/local/include/FL/images -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -D_THREAD_SAFE -D_REENTRANT -o chapter.12.7.9-2.o -c chapter.12.7.9-2.cpp
  - g++ -o chapter.12.7.9-2.exe -L"../GUI" chapter.12.7.9-2.o -lstdc++ -lbookgui -lfltk -lfltk_images /usr/local/lib/libfltk.a -lfltk_jpeg –lpthread

Next lecture

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

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!


BY


Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης