DM560

Introduction to Programming in C++

Graphical Interface
Object Oriented Programming

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

Overview

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

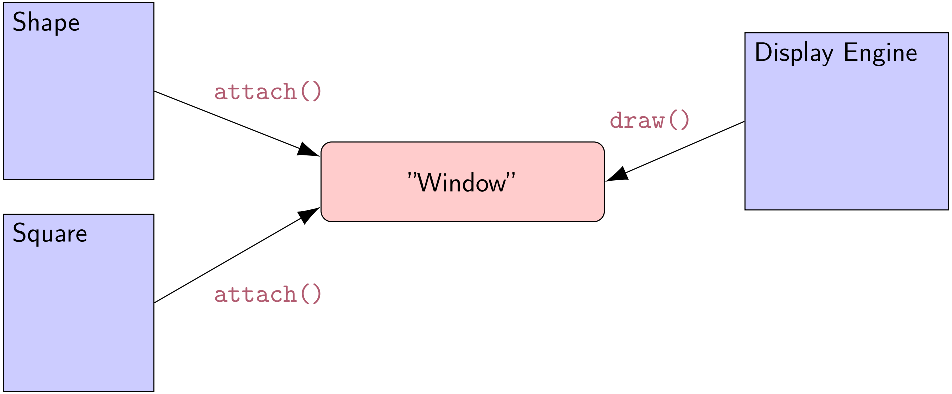
Outline

Motivation

Why bother with graphics and Graphical User Interface (GUI)?

- It's very common to need it if you write conventional PC applications
- It's useful
    - Instant feedback
    - Graphing functions
    - Displaying results
- It can illustrate some generally useful concepts and techniques
- It can only be done well using some pretty neat language features
- Lots of good (small) code examples
- It can be non-trivial to "get"the key concepts, thus we devote some lectures to it
- Graphics is fun!

Display Model



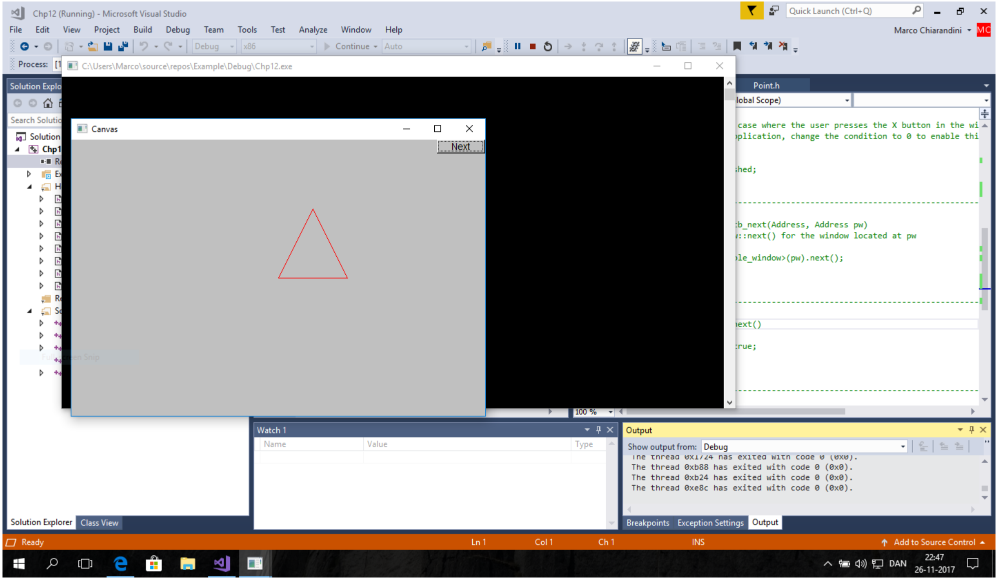
- Objects (such as graphs) are attached to a window.
- The display engine invokes display command (such as "draw line from x to y") for the objects in a window
- Objects such as Square contain vectors of lines, text, etc. for the window to draw

Display Model

An example illustrating the display model

```
int main()
{
   using namespace Graph_lib;   // use our graphics interface library

   Point tl(100,200);                     // a point

   Simple_window win(tl,600,400,"Canvas");      // make a simple window

   Polygon poly;                // make a shape (a polygon)

   poly.add(Point(300,200));    // add three points to the polygon
   poly.add(Point(350,100));
   poly.add(Point(400,200));

   poly.set_color(Color::red);        // make the polygon red

   win.attach(poly);            // connect poly to the window

   win.wait_for_button();       // give control to the display engine
}
```

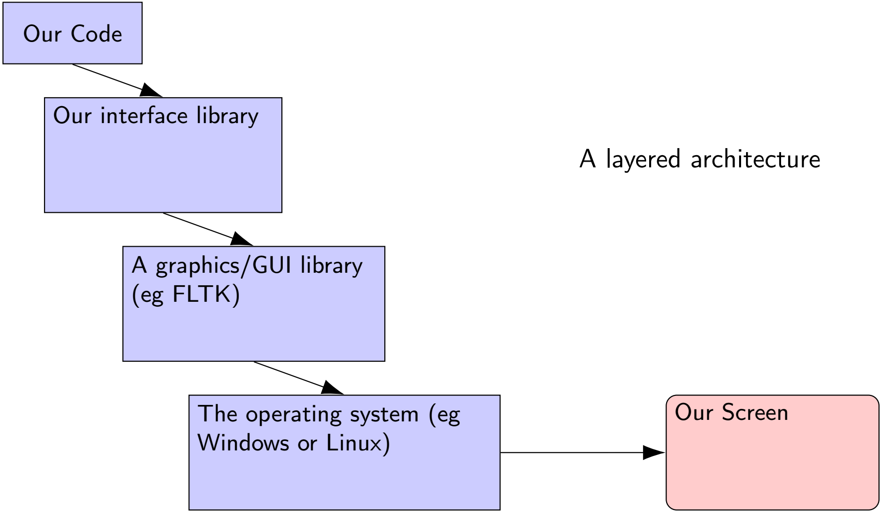
The Resulting Screen



Graphics/GUI libraries

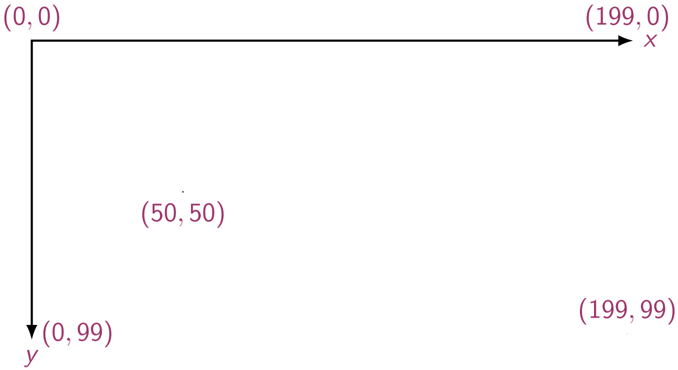
- We will be using a few interface classes wrote by Bjarne Stroustrup
  - Interfacing to a popular GUI toolkit: Fast Light Tool Kit (FLTK) `www.fltk.org`
  - Installation: try following Appendix D and ask teacher/instructor/friend
    `https://bewuethr.github.io/installing-fltk-133-under-visual-studio/`
    FLTK, the GUI and graphics classes from `common`, Project settings
- This model is far simpler than common toolkit interfaces
  - The FLTK (very terse) documentation is 370 pages
  - Our interface library is <20 classes and <500 lines of code
  - You can write a lot of code with these classes and you can build more classes on them
- The code is portable (Windows, Unix, Mac, etc.)
- This model extends to most common graphics and GUI uses
- The general ideas can be used with any popular GUI toolkit Once you understand the graphics classes you can easily learn any GUI/graphics library
  Well, relatively easily – these libraries are huge (eg, Qt libraries)

Graphics/GUI libraries



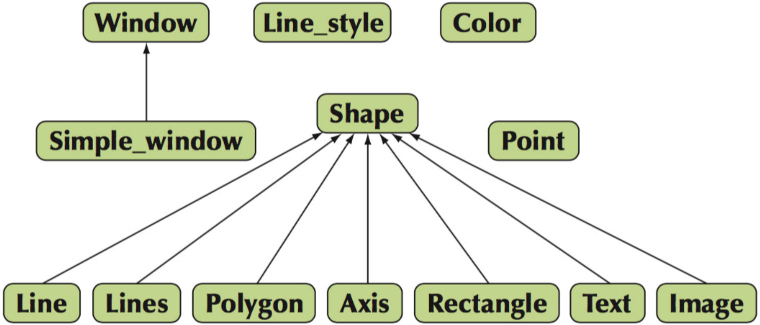
A layered architecture

Coordinates



- Oddly, y-coordinates grow downwards $\rightsquigarrow$ (right, down)
- Coordinates identify pixels in the window on the screen
- You can resize a window (changing `x_max()` and `y_max()`)

Interface Classes



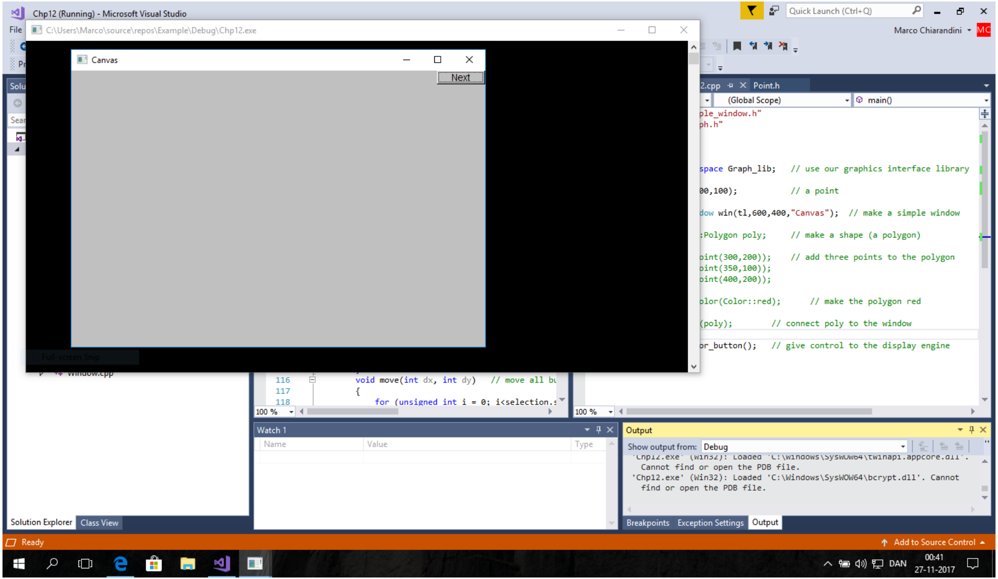
- An arrow ⟶ means "is a kind of"
- Color, Line_style, and Point are utility classes used by the other classes
- Window is our interface to the GUI library (which is our interface to the screen)
- Extensible: Grid, Block_chart, Pie_chart, etc.
- Later, GUI: Button, In_box, Out_box, ...

Demo Code 1

```
// Getting access to the graphics system (don't forget to install):
#include "Simple_window.h"       // stuff to deal with your system's windows
#include "Graph.h"               // graphical shapes

using namespace Graph_lib;       // make names available
```

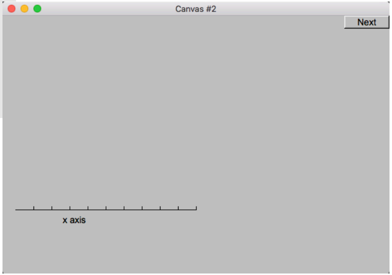
```
// in main():

Simple_window win(Point(100,100),600,400,"Canvas");
                // screen coordinate (100,100) is top left corner of window
                // window size(600 pixels wide by 400 pixels high)
                // title: Canvas
win.wait_for_button();  // Display!
```

A "Blank Canvas"



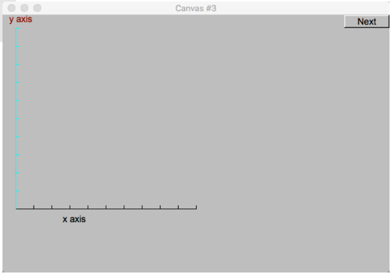
Demo Code 2 — Add an X-Axis

```
Axis xa(Axis::x, Point(20,300), 280, 10, "x axis");
  // make an Axis
        // an axis is a kind of Shape
        // Axis::x means horizontal
        // starting at (20,300)
        // 280 pixels long
        // 10 "notches" ("tick marks")
        // text "x axis"
win.set_label("Canvas #2");
win.attach(xa); // attach axis xa to the window
win.wait_for_button();
```



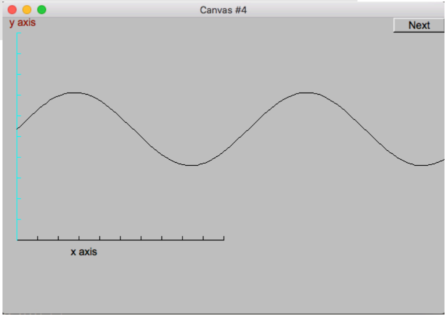
Demo Code 3 — Add a Y-Axis

```
win.set_label("Canvas #3");

Axis ya(Axis::y, Point(20,300), 280, 10, "y axis");
ya.set_color(Color::cyan); // a color for the axis
ya.label.set_color(Color::dark_red); // for the text

win.attach(ya);
win.wait_for_button();
```



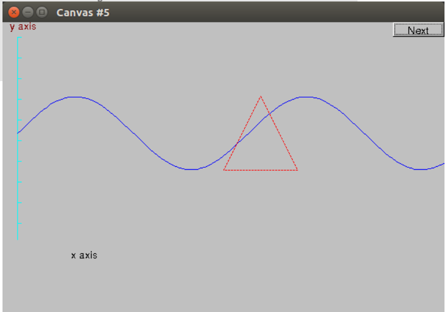
Demo Code 4 — Add a Sine Curve

```
win.set_label("Canvas #4");

Function sine(sin,0,100,Point(20,150),1000,50,50);      // sine curve
                // plot sin() in the range [0:100)
                // with (0,0) at (20,150)
                // using 1000 points
                // scale x values *50, scale y values *50

win.attach(sine);
win.wait_for_button();
```



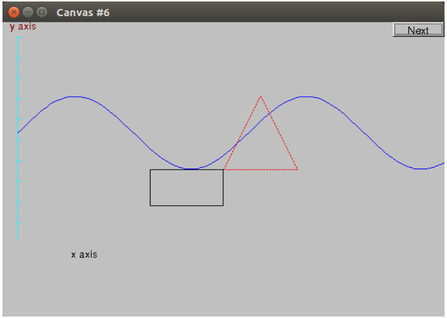
Demo Code 5 — Color Curve and Add a Triangle

```
win.set_label("Canvas #5");

sine.set_color(Color::blue); // I changed my mind about sine's color

Polygon poly;  // make a polygon (a kind of Shape)
poly.add(Point(300,200)); // three points make a triangle
poly.add(Point(350,100));
poly.add(Point(400,200));

poly.set_color(Color::red); // change the color
poly.set_style(Line_style::dash); // change the line style

win.attach(poly);
win.wait_for_button();
```



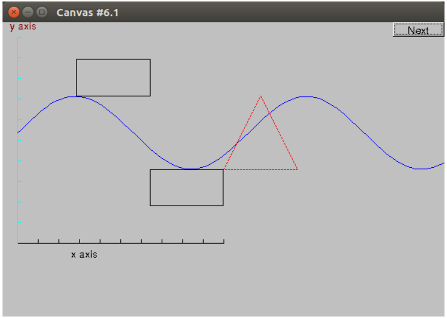
Demo Code 6 — Add a Rectangle

```
win.set_label("Canvas #6");

Rectangle r(Point(200,200), 100, 50);    // top left point, width, height

win.attach(r);
win.wait_for_button();
```



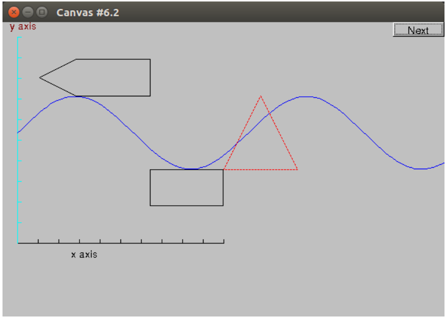
Demo Code 6.1 — Add a Shape like a Rectangle

```
Closed_polyline poly_rect;
poly_rect.add(Point(100,50));
poly_rect.add(Point(200,50));
poly_rect.add(Point(200,100));
poly_rect.add(Point(100,100));

win.attach(poly_rect);

win.set_label("Canvas #6.1");
```



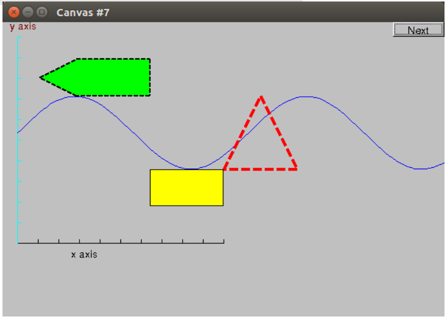
Demo Code 6.2 — Add a Point to Polygon

```
poly_rect.add(Point(50,75));     // now poly_rect has 5 points

win.set_label("Canvas #6.2");
```



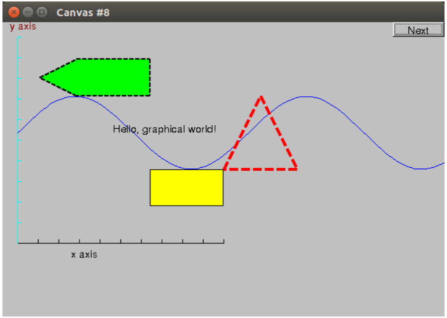
Demo Code 7 — Add Fill

```
r.set_fill_color(Color::yellow); // color the inside of the rectangle

poly.set_style(Line_style(Line_style::dash,4));
// make the triangle fat

poly_rect.set_fill_color(Color::green);
poly_rect.set_style(Line_style(Line_style::dash,2));

win.set_label("Canvas #7");
```



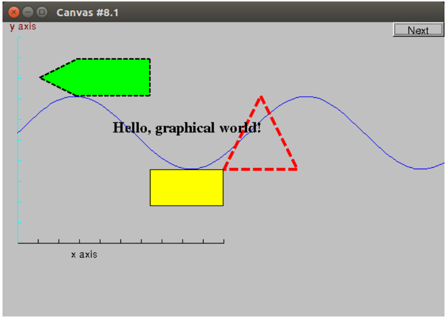
Demo Code 8 — Add Text

```
Text t(Point(150,150),"Hello, graphical world!");   // add text
                 // point is lower left corner on the baseline
win.attach(t);

win.set_label("Canvas #8");
```



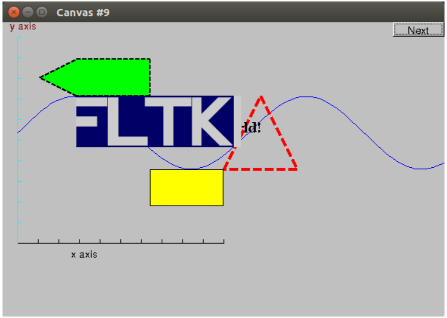
Demo Code 8.1 — Modify Text Font and Size

```
t.set_font(Graph_lib::Font::times_bold);
t.set_font_size(20);  // height in pixels

win.set_label("Canvas #8.1");
```



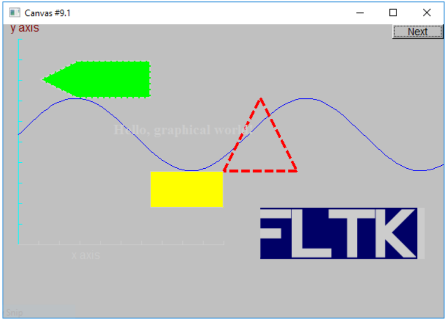
Demo Code 9 — Add an Image

```
Image ii(Point(100,100),"Resources/fltk.gif"); // open an image file
win.attach(ii);
win.set_label("Canvas #9");
```



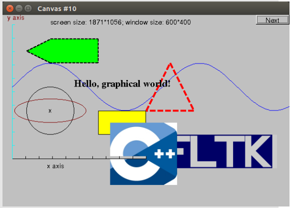
Demo Code 9.1 — Move the Image

```
ii.move(250,150);   // move 250 pixels to the right (-250 moves left)
                    // move 150 pixels down (-150 moves up)

win.set_label("Canvas #9.1");
win.wait_for_button();
```



Demo Code 10 — Add Shapes, More Text

```
Circle c(Point(100,200),50); // center, radius
win.attach(c);

Ellipse e(Point(100,200), 75,25); // center, horizontal radius, vertical radius
e.set_color(Color::dark_red);
win.attach(e);

Mark m(Point(100,200),'x');
win.attach(m);

ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes(Point(100,20),oss.str());
win.attach(sizes);

Image cal(Point(225,225), "Resources/0603_sdt-cpp.jpeg"); // 200*220 pixel jpeg
cal.set_mask(Point(40,50),140,130); // display center of image
win.attach(cal);

win.set_label("Canvas #10");
win.wait_for_button();
```



Boilerplate

Bopilerplate standardized piece of code for use in a computer program

```
#include "Graph.h"           // header for graphs
#include "Simple_window.h"   // header containing window interface

int main ()
try
{
  // the  main part of your code
}
catch(exception& e) {
  cerr << "exception: " << e.what() << '\n';
  return 1;
}
catch (...) {
  cerr << "Some exception\n";
  return 2;
}
```

Primitive and Algorithms

- The demo shows the use of library primitives
  - Just the primitives
  - Just the use
- Typically what we display is the result of
  - an algorithm
  - reading data
- Next content:
  - 13: Graphics Classes
  - 14: Graphics Class Design
  - 15: Graphing Functions and Data
  - 16: Graphical User Interfaces

Outline

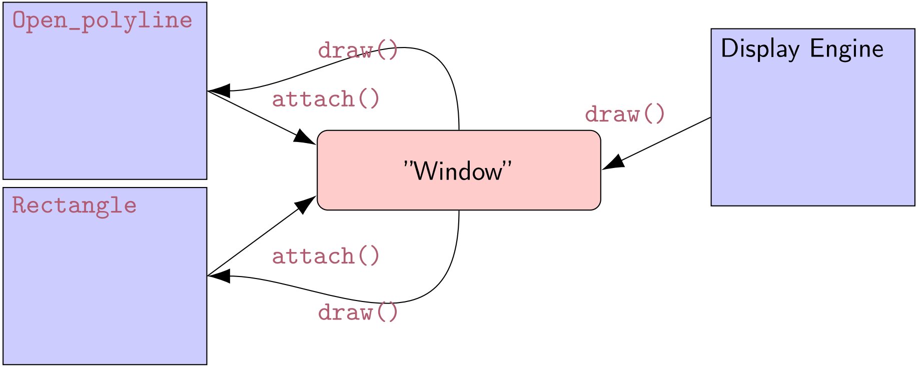
Overview

We learn how the shapes and operations of the previous section are actually implemented

- Graphing
    - Model
    - Code organization

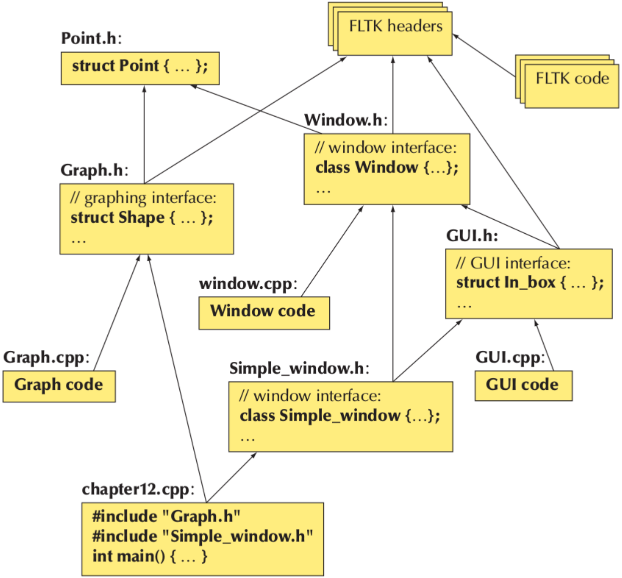
- Interface classes
    - Point
    - Line
    - Lines
    - Grid
    - Open Polylines
    - Closed Polylines
    - Color
    - Text
    - Unnamed objects

Display Model



- Objects (such as graphs) are attached to (placed in) a window.
- The display engine invokes display command (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

- `.h` (header file)
  - 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
- You can read the `Graph.h` header and later the `Graph.cpp` implementation file
- Instead, `Window.h` header and the `Window.cpp` implementation file are heavy 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 (details in Chapter 14)
- **Color** – as you see it on the screen

class vs struct

As from the Cpp Core Guidelines

From a language perspective `class` and `struct` differ only in the default visibility of their members. (In `class` it is `private`; in `struct` it is `public`.)

C.1: Organize related data into structures (`struct`s or `class`es)

```
void draw(int x, int y, int x2, int y2); // BAD: unnecessary implicit relationshi
void draw(Point from, Point to);          // better
```

C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently
An invariant is a logical condition for the members of an object that a constructor must establish for the public member functions to assume.

C.8: Use class rather than struct if any member is non-public

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!
```

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
};
```

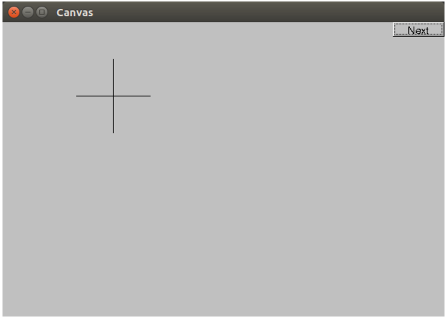
Terminology:

- 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!
```



Implementation: Lines

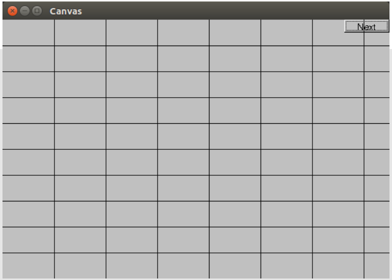
```
void Lines::add(Point p1, Point p2)      // use Shape's add()
{
  Shape::add(p1);
  Shape::add(p2);
}

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)  // 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)
```

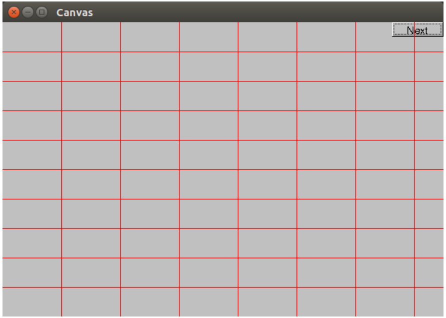
Oops! Last column is narrow, there's a grid line on top of the Next button, etc.— tweaking required (as usual)

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;
};
```

Example: Draw Red Grid

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

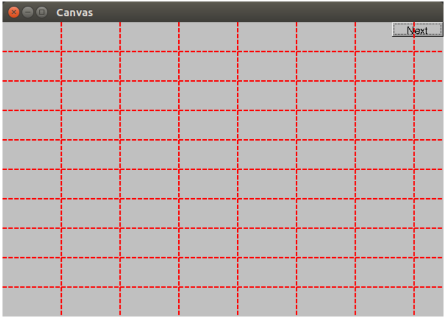


Line_style

```
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));
```

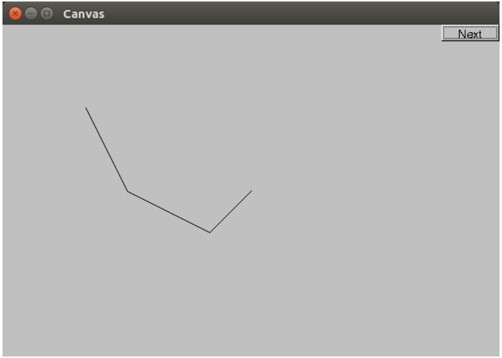


Polylines

```
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?)
};
```

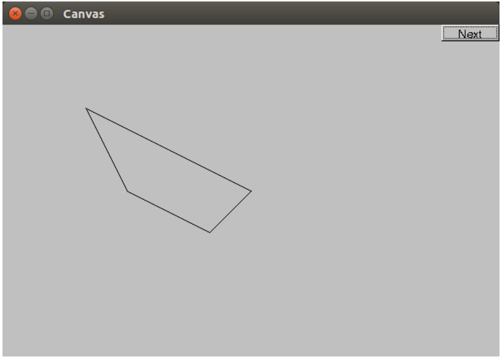
Open_polyline

```
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));
```

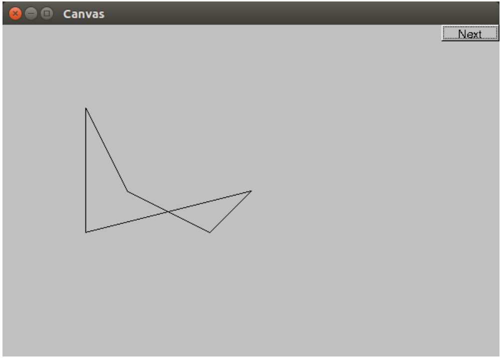


Closed_polyline

```
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

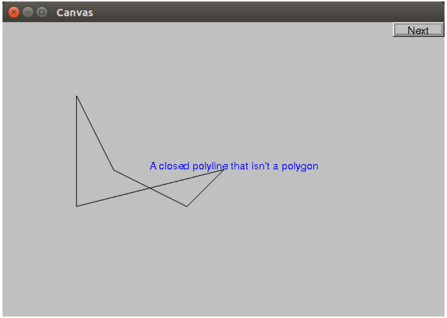


Text

```
struct Text : Shape {
  Text(Point x, const string& s)        // x is the bottom left of the first letter
       : lab(s),
         fnt(fl_font()),         // default character font
         fnt_sz(fl_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

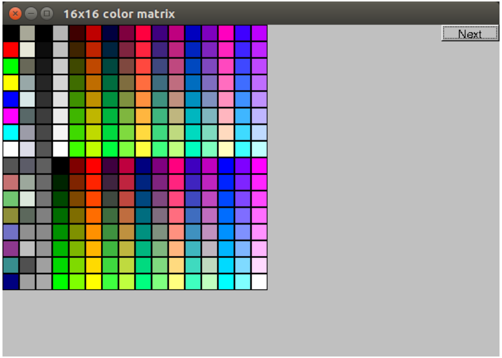
```
void Text::draw_lines() const
{
  fl_draw(lab.c_str(), point(0).x, point(0).y);
}

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

Color Matrix

Drawing a color matrix.

Good example of:

- how many colors we have to work with
- how messy two-dimensional addressing can be (see Matrices chp 24)
- how to avoid inventing names of hundreds of objects



Color Matrix (16×16)

```
Simple_window win20(Point(100,100),600,400,"16x16 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
```

Outline

Overview

- Library design considerations
- Class hierarchies (object-oriented programming)
- Data hiding

Ideals

Our ideal of program design is to represent the concepts of the application domain directly in code.

If you understand the application domain, you understand the code, and vice versa. For example:

- `Window` – a window as presented by the operating system
- `Line` – a line as you see it on the screen
- `Point` – a coordinate point
- `Color` – as you see it on the screen
- `Shape` – what's common for all shapes in our Graph/GUI view of the world

In the last example, `Shape` is different from the rest in that it is a generalization. You can't make an object that is "just a Shape"

Logically Identical Operations Have Same Name

For every class:

- `draw_lines()` does the drawing
- `move(dx,dy)` does the moving
- `s.add(x)` adds some x (e.g., a point) to a shape s.

For every property `x` of a `Shape`,

- `x()` gives its current value and
- `set_x()` gives it a new value

Example:

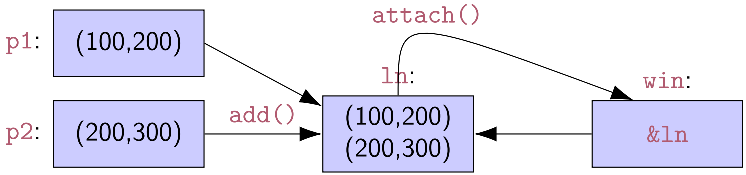
```
Color c = s.color();
s.set_color(Color::blue);
```

Logically Different Operations Have Different Names

```
Lines ln;
Point p1(100,200);
Point p2(200,300);
ln.add(p1,p2);                       // add points to ln (make copies)
win.attach(ln);                      // attach ln to window
```

Why not `win.add(ln)`?

`add()` copies information; `attach()` just creates a reference we can change a displayed object after attaching it, but not after adding it



Expose Uniformly

Data should be `private`

- Data hiding – so it will not be changed inadvertently
- Use `private` data, and pairs of public access functions to **get** and **set** the data

```
c.set_radius(12);    // set radius to 12
c.set_radius(c.radius()*2); // double the radius (fine)
c.set_radius(-9);           // set_radius() could check for negative,
                            // but doesn't yet
double r = c.radius();      // returns value of radius
c.radius = -9;           // error: radius is a function (good!)
c.r = -9;                // error: radius is private (good!)
```

Our functions can be private or public

- `public` for interface
- `private` for functions used only internally to a class

What Does `private` Imply?

- We can change our implementation after release
- We don't expose FLTK types used in representation to our users
  We could replace FLTK with another library without affecting user code
- We could provide checking in access functions
  But we haven't done so systematically (later?)
- Functional interfaces can be nicer to read and use
  E.g., `s.add(x)` rather than `s.points.push_back(x)`
- We enforce immutability of shape
  - Only color and style change; not the relative position of points
  - `const` member functions
- The value of this encapsulation varies with application domains
  - Is often most valuable
  - Is the ideal, i.e., hide representation unless you have a good reason not to

Interface Design
Regular Interfaces

```
Line ln(Point(100,200),Point(300,400));
Mark m(Point(100,200), 'x');      // display a single point as an 'x'
Circle c(Point(200,200),250);

// Alternative (not supported):
Line ln2(x1, y1, x2, y2);                  // from (x1,y1) to (x2,y2)

// How about? (not supported):
Rectangle s1(Point(100,200),200,300);          // width==200 height==300
Rectangle s2(Point(100,200),Point(200,300));   // width==100 height==100

Rectangle s3(100,200,200,300);// is 200,300 a point  or a width plus a height?
```

A Library

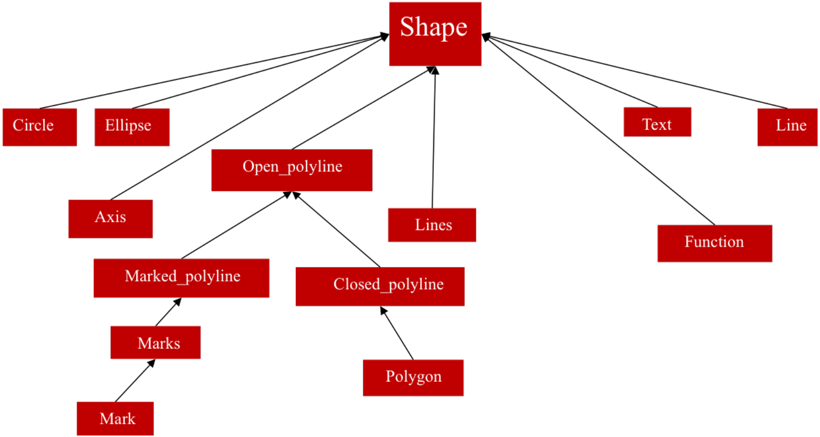
- A collection of classes and functions meant to be used together As building blocks for applications To build more such "building blocks“

- A good library models some aspect of a domain
  - It doesn’t try to do everything
  - Our library aims at simplicity and small size for graphing data and for very simple GUI

- We can’t define each library class and function in isolation
  - A good library exhibits a uniform style (regularity)

Class Shape

All our shapes are based on the Shape class
E.g. a Polygon is a kind of Shape



Class Shape is Abstract

We can't make a "plain" Shape

```
protected:
    Shape();        // protected to make class Shape abstract
```

For example:

```
Shape ss;    // error: cannot construct Shape
```

Protected means "can only be used from this class or from a derived class"

Instead, we use Shape as a base class

```
struct Circle : Shape { // "a Circle is a Shape"
          // ...
};
```

Class `Shape`

- `Shape` ties our graphics objects to "the screen"
  - Window "knows about" Shapes
  - All our graphics objects are kinds of `Shape`s

- `Shape` is the class that deals with color and style
  It has `Color` and `Line_style` members

- `Shape` can hold Points

- `Shape` has a basic notion of how to draw lines
  It just connects its Points

Class Shape

Shape deals with color and style

It keeps its data private and provides access functions

```
  void set_color(Color col);
  Color color() const;
  void set_style(Line_style sty);
  Line_style style() const;
  // ...
private:
  // ...
  Color line_color;
  Line_style ls;
```

Class Shape

Shape stores Points

It keeps its data private and provides access functions

```
    Point point(int i) const;   // read-only access to points
    int number_of_points() const;
    // ...
protected:
    void add(Point p);   // add p to points
    // ...
private:
    vector<Point> points;   // not used by all shapes
```

Class Shape

- Shape itself can access points directly:

```
void Shape::draw_lines() const  // draw connecting lines
{
    if (color().visible() && points.size()>1)
        for (int i=1; i<points.size(); ++i)
            fl_line(points[i-1].x,points[i-1].y,points[i].x,points[i].y);
}
```

- Others (incl. derived classes) use point() and number_of_points(). Why?

```
void Lines::draw_lines() const  // draw a line for each pair of points
{
    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);
}
```

Class Shape

Implementation of Drawing

```
void Shape::draw() const
    // The real heart of class Shape (and of our graphics interface system)
    // called by Window (only)
{
    Fl_Color oldc = fl_color();  // save old color
    // there is no good portable way of retrieving the current style (sigh!)
    fl_color(line_color.as_int());          // set color and style
    fl_line_style(ls.style(),ls.width());

    draw_lines();       // call the appropriate draw_lines()
                                             // a virtual call
                                             // here is what is specific for a "derived class" is c

    fl_color(oldc);        // reset color to previous
    fl_line_style(0);      // (re)set style to default
}
```

Class Shape

- In class Shape

```
virtual void draw_lines() const;      // draw the appropriate lines
```

- In class Circle

```
void draw_lines() const { /* draw the Circle */ }
```

- In class Text

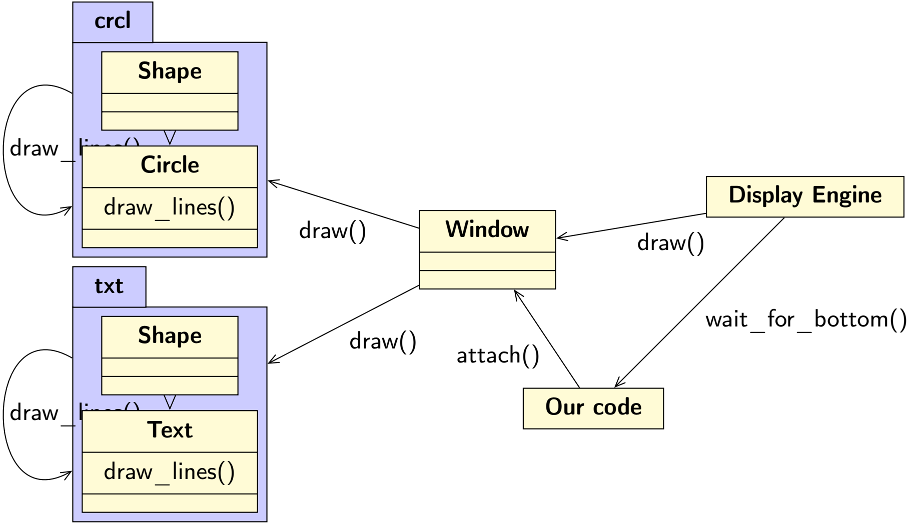
```
void draw_lines() const { /* draw the Text */ }
```

- Circle, Text, and other classes:
  - Derive from Shape
  - May override draw_lines()

Class Shape

```
class Shape  {  // deals with color and style, and holds a sequence of lines
public:
   void draw() const;             // deal with color and call draw_lines()
   virtual void move(int dx, int dy);   // move the shape +=dx and +=dy

   void set_color(Color col);     // color access
   int color() const;
   // ... style and fill_color access functions ...

   Point point(int i) const;      // (read-only) access to points
   int number_of_points() const;
protected:
   Shape();                       // protected to make class Shape abstract
   void add(Point p);             // add p to points
   virtual void draw_lines() const;     // simply draw the appropriate lines
private:
   vector<Point> points;                // not used by all shapes
   Color  lcolor;                       // line color
   Line_style  ls;                      // line style
   Color  fcolor;                       // fill color
   // ... prevent copying ...
};
```

Display Model Completed



Language Mechanisms

Most popular definition of **object-oriented programming**:

$$\text{OOP} \equiv \text{inheritance} + \text{polymorphism} + \text{encapsulation}$$

- Inheritance: Base and derived classes

```
struct Circle : Shape { ... };
```

- Polymorphism: Also called run-time polymorphism or dynamic dispatch Virtual functions

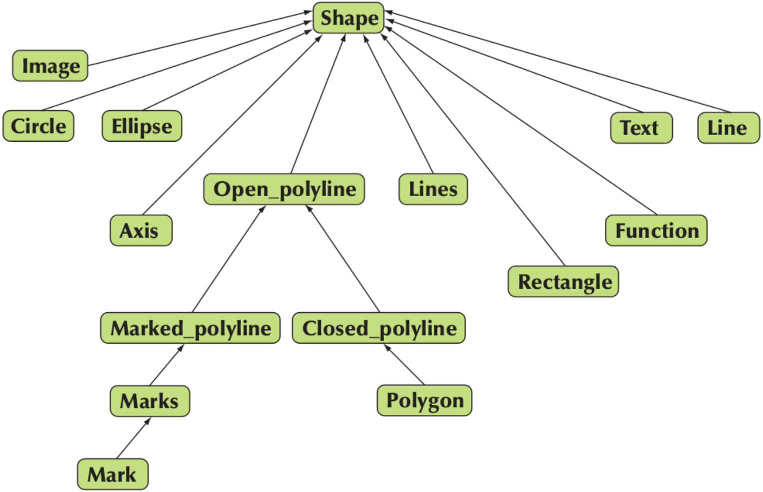
```
virtual void draw_lines() const;
```

- Encapsulation: Private and protected

```
protected: Shape();
private: vector<Point> points;
```

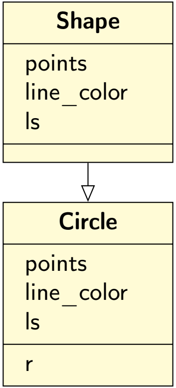
A Simple Class Hierarchy

We design a simple (and mostly shallow) class hierarchy based on `Shape`



Object Layout

The data members of a derived class are simply added at the end of its base class (a `Circle` is a `Shape` with a radius)

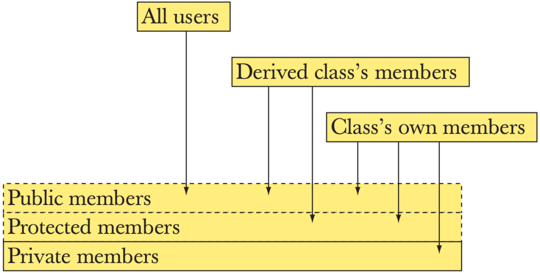


Benefits of Inheritance

- Interface inheritance
  - A function expecting a shape (a `Shape&`) can accept any object of a class derived from `Shape`.
  - Simplifies use (sometimes dramatically)
  - We can add classes derived from Shape to a program without rewriting user code
    (Adding without touching old code is one of the "holy grails" of programming)
- Implementation inheritance
  - Simplifies implementation of derived classes
  - Common functionality can be provided in one place
  - Changes can be done in one place and have universal effect
    (Another "holy grail")

Access Model

A member of a class (data, function, or type member) can be:
`private`, `protected`, or `public`



If a base class of a derived class `D` is

- `private`, then its `public` and `protected` members can be accessed only by members of `D`
- `protected`, then its `public` and `protected` members can be accessed only by members of `D` and of classes derived from `D`
- `public`, then its `public` members can be accessed by all

Pure Virtual Functions

Often, a function in an interface can't be implemented
E.g. the data needed is "hidden" in the derived class

- Make it a pure virtual function (=0)
- We must ensure that a derived class implements that function

Abstract interfaces (pure interfaces, abstract classes): classes that cannot be instantiated and only used as base classes:

- contain pure virtual functions
- have protected constructors.

```
struct Engine { // interface to electric motors
  // no data
  // (usually) no constructor
  virtual double increase(int i) =0;    // must be defined in a derived class
  // ...
  virtual ~Engine();    // (usually) a virtual destructor
};
```

```
Engine eee;     // error: Collection is an abstract class
```

Pure Virtual Functions

A pure interface can be used as a base class
(Constructors and destructors are described in detail in chapters 17-19)

```
Class M123 : public Engine { // engine model M123
  // representation
public:
  M123();        // construtor: initialization, acquire resources
  double increase(int i) { /* ... */ }  // overrides Engine::increase
  // ...
  ~M123();       // destructor: cleanup, release resources
};
```

```
M123 window3_control;   // OK
```

Technicality: Copying

If you don't know how to copy an object, prevent copying
Abstract classes typically should not be copied

```
class Shape {
  // ...
  Shape(const Shape&) = delete;                 // don't ``copy construct''
  Shape& operator=(const Shape&) = delete;      // don't ``copy assign''
};
```

```
void f(Shape& a)
{
  Shape s2 = a; // error: no Shape ``copy constructor'' (it's deleted)
  a = s2;       // error: no Shape ``copy assignment'' (it's deleted)
}
```

Technicality: Overriding

To override a virtual function, you need

- A virtual function
- Exactly the same name
- Exactly the same type

```
struct B {
  void f1();     // not virtual
  virtual void f2(char);
  virtual void f3(char) const;
  virtual void f4(int);
};

struct D : B {
  void f1();             // doesn't override
  void f2(int);          // doesn't override
  void f3(char);         // doesn't override
  void f4(int); // overrides
};
```

Technicality: Overriding

To override a virtual function, you need

- A virtual function
- Exactly the same name
- Exactly the same type

```
struct B {
  void f1();      // not virtual
  virtual void f2(char);
  virtual void f3(char) const;
  virtual void f4(int);
};

struct D : B {
  void f1() override;           // error
  void f2(int) override;        // error
  void f3(char) overrride;      // error
  void f4(int) override;        // ok
};
```

Technicality: Overriding

To invoke a virtual function, you need

- a reference, or
- a pointer

```
D d1;
B& bref = d1;    // d1 is a D, and a D is a B, so d1 is a B
bref.f4(2);      // calls D::f4(2) on d1 since bref names a D

// pointers are in chapter 17
B *bptr = &d1;  // d1 is a D, and a D is a B, so d1 is a B
bptr->f4(2);            // calls D::f4(2) on d1 since bptr points to a D
```

Summary

1. A Graphical Interface
2. Graphics Classes
3. Graph Class Design