Chapter 14
Graph class design

Bjarne Stroustrup

www.stroustrup.com/Programming

Abstract

- We have discussed classes in previous lectures
- Here, we discuss design of classes
  - 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
- The last example, **Shape**, is different from the rest in that it is a generalization.
  - You can't make an object that's "just a Shape"

Logically identical operations have the same name

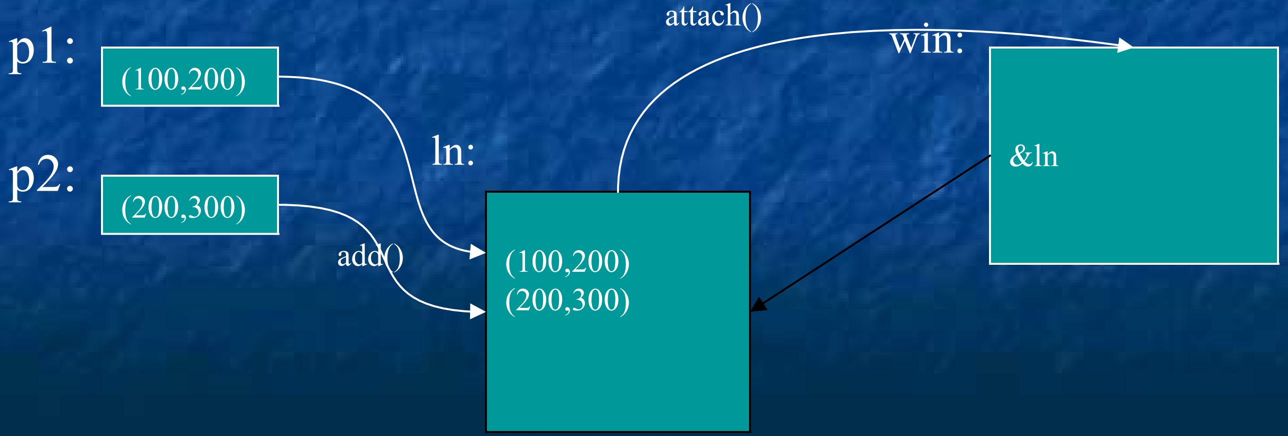
- For every class,
  - **draw_lines()** does the drawing
  - **move(dx,dy)** does the moving
  - **s.add(x)** adds some **x** (*e.g.*, a point) to a shape **s**.
- For every property **x** of a Shape,
  - **x()** gives its current value and
  - **set_x()** gives it a new value
  - e.g.,

```
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” buy us?

- 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

“Regular” interfaces

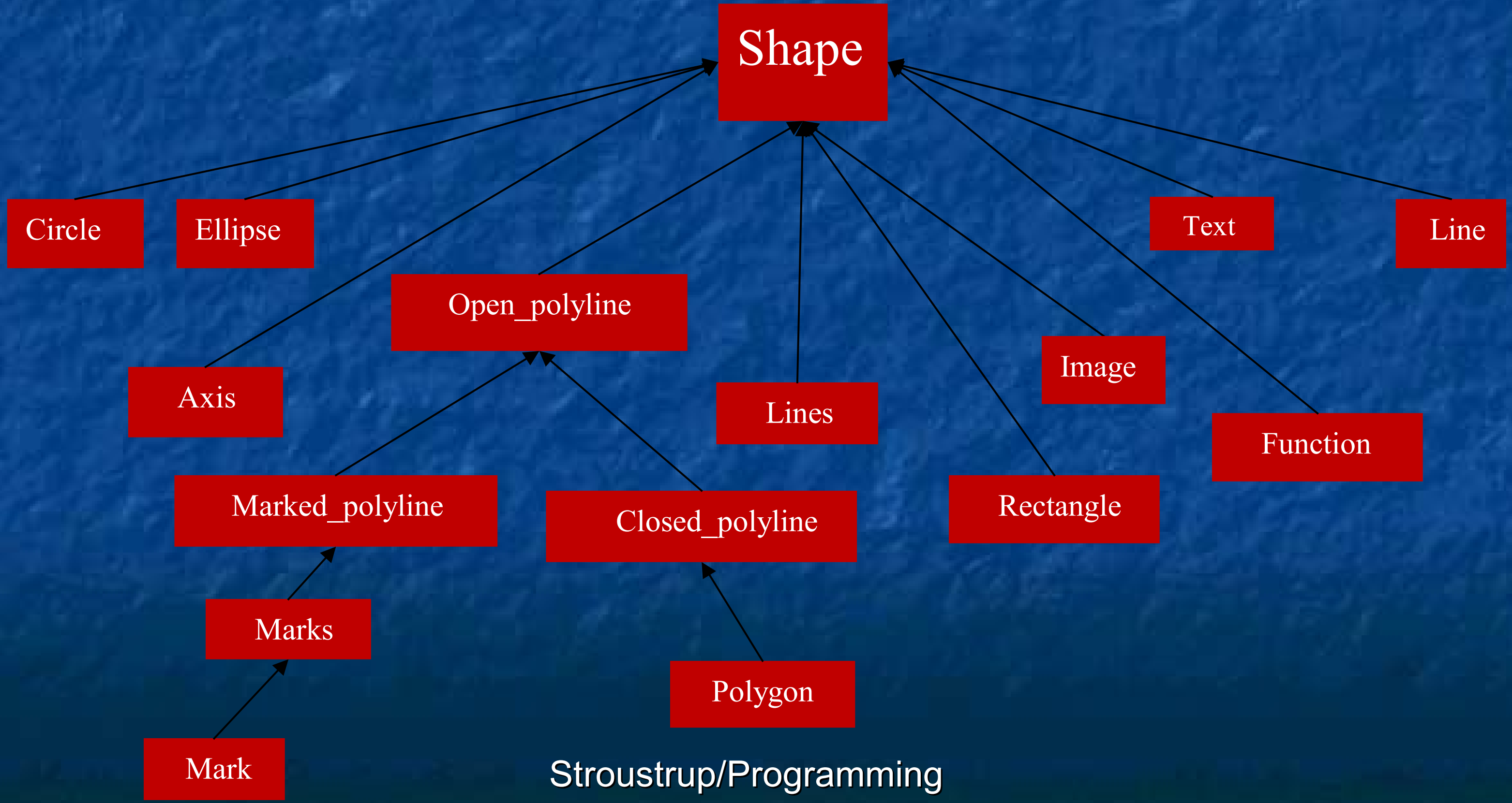
```
Line ln(Point(100,200),Point(300,400));
Mark m(Point(100,200), 'x');        // display a single point as an 'x'
Circle c(Point(200,200),250);

// 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

- You 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 **Shapes**
- **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() && 1<points.size())
    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 (basic idea of drawing)

```
void Shape::draw() const
    // The real heart of class Shape (and of our graphics interface system)
    // called by Window (only)
{
    // … save old color and style …
    // … set color and style for this shape…

    // … draw what is specific for this particular shape …
    // … Note: this varies dramatically depending on the type of shape …
    // … e.g. Text, Circle, Closed_polyline

    // … reset the color and style to their old values …
}
```

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 done

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

Note!

Class shape

- In class **Shape**

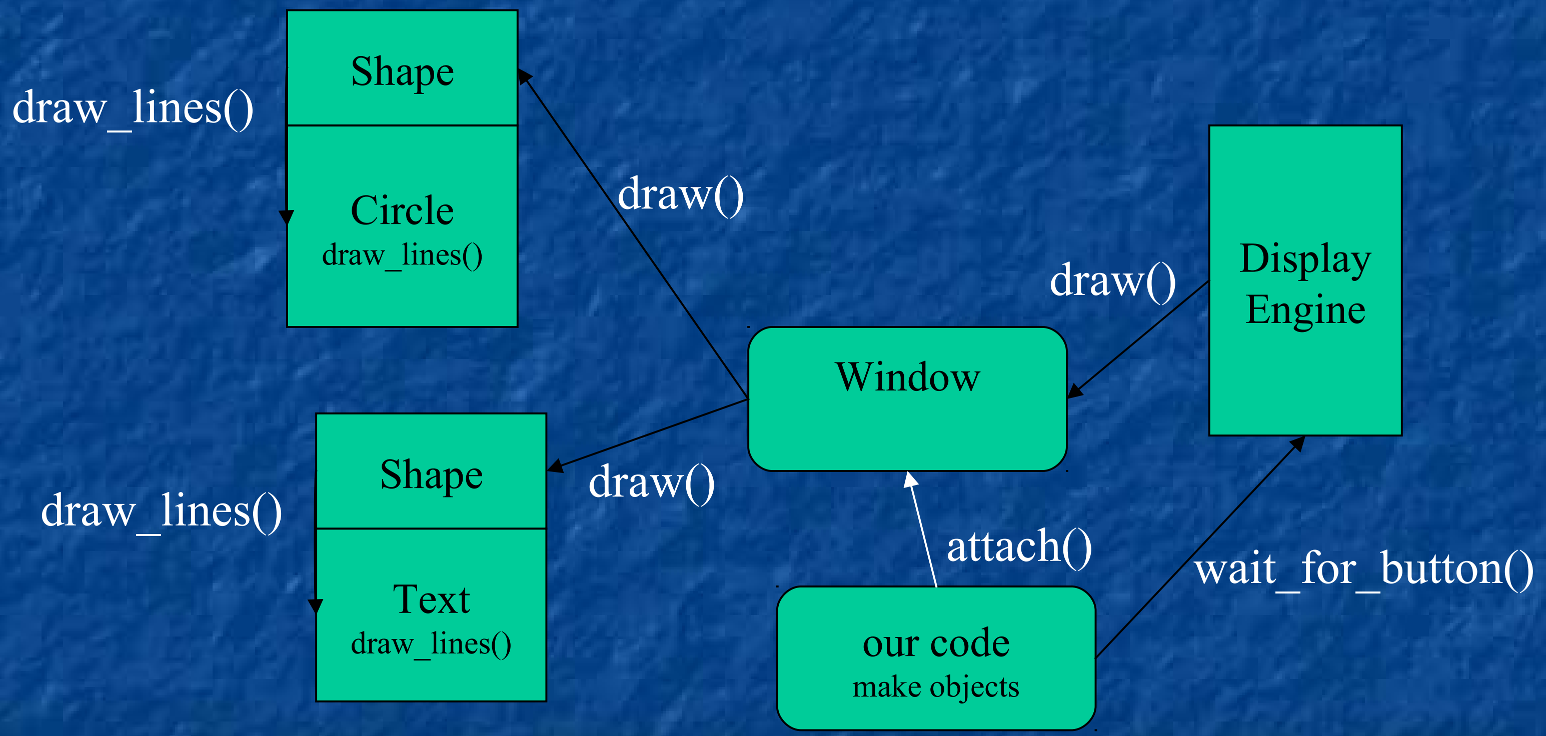
  virtual void draw_lines() const; *// draw the appropriate lines*
- In class **Circle**

  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  {          // 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

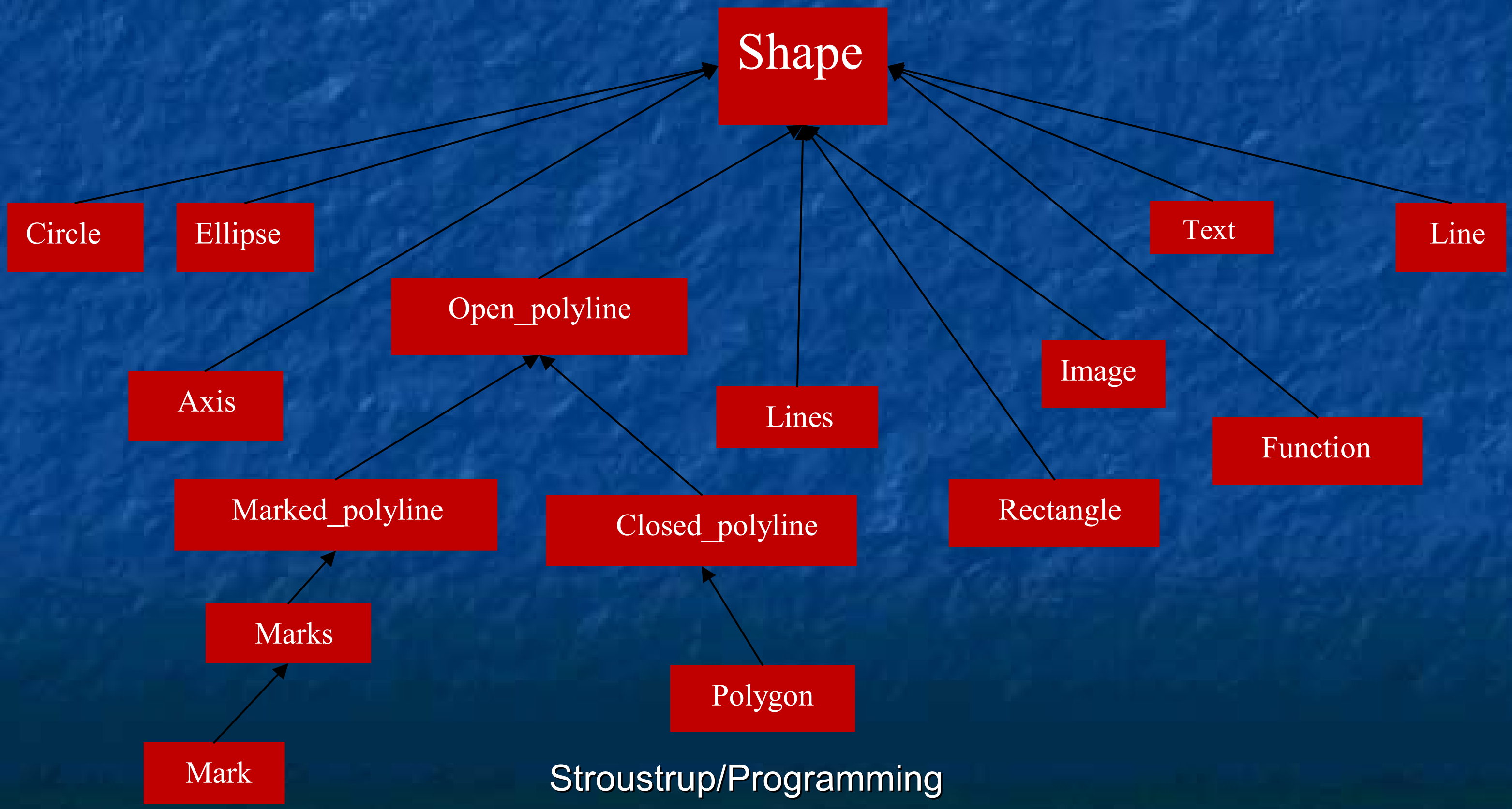
- Most popular definition of object-oriented programming:

  OOP == inheritance + polymorphism + encapsulation

- Base and derived classes // *inheritance*
  - **struct Circle : Shape { … };**
  - Also called "inheritance"
- Virtual functions // *polymorphism*
  - **virtual void draw_lines() const;**
  - Also called "run-time polymorphism" or "dynamic dispatch"
- Private and protected // *encapsulation*
  - **protected: Shape();**
  - **private: vector<Point> points;**

A simple class hierarchy

- We chose to use 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)

Shape:

points
line_color
ls

Circle:

points

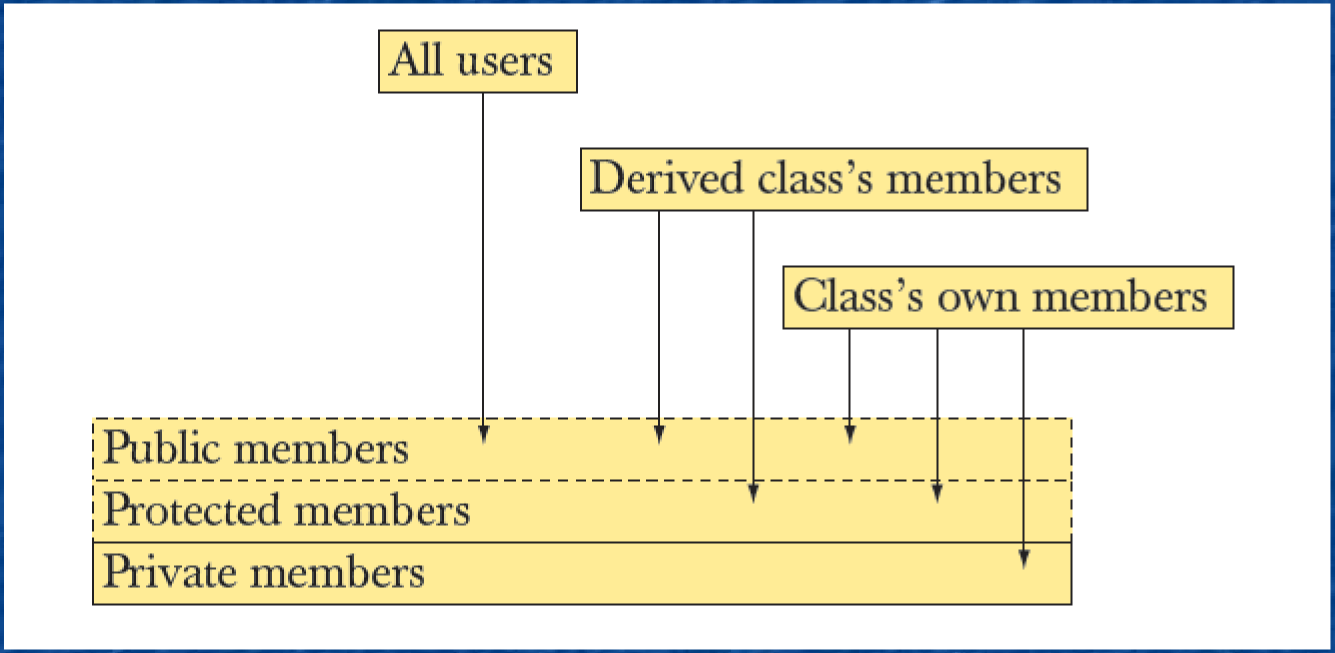
line_color
ls
----
r

Benefits of inheritance

Parasol
Smarter computing.
Texas A&M University

- 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 (data, function, or type member) or a base can be
  - Private, protected, or public

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (=0)
- This is how we define truly abstract interfaces ("pure interfaces")

```
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 then be used as a base class
  - Constructors and destructors will be describe d 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)
}
```

Prevent copying C++98 style

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

```
class Shape {
    // …
private:
    Shape(const Shape&);        // don't "copy construct"
    Shape& operator=(const Shape&); // don't "copy assign"
};

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

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) override;       // 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
```

Next lecture

- Graphing functions and data