USTC

Chapter 14
Graph class design

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 上一章我重点讨论了各种实际使用的具体图形类
- 现在，我们开始讨论类的相关设计问题
  - 库设计相关思考
  - 类的分层关系 (object-oriented programming)
  - 数据隐藏

原则

- 程序设计的原则是直接用代码**精确地**表述应用领域的概念
  - 如果你很好地理解了应用领域，你就能理解代码，反之亦然，例如：
    - Window – 操作系统表示的一个窗口
    - Line – 一条线，如你在屏幕中所见
    - Point – 一个坐标点
    - Color – 颜色，如你在屏幕中所见
    - Shape – 所有形状的统称（以我们的图形/GUI视角来看待世界时）
- Shape, 与其它具体的图形类不同，它是一种一般化的概念.
  - 不能生成一个Shape对象

逻辑上相同的操作/函数采用相同的名称！

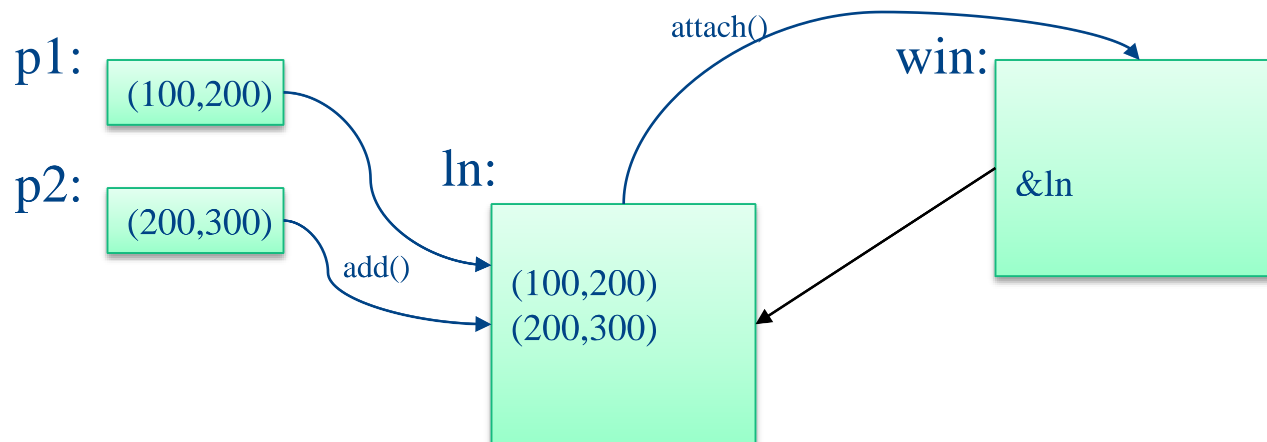
- 等价的函数尽可能
  - 有相同的函数名
  - 接受相同的参数，及相同的参数顺序
- 对所有的类：
  - **draw_lines()** 执行绘制功能
  - **move(dx,dy)** 执行移动功能
  - **s.add(x)** 添加某个 **x** (*e.g.*, a point) 到一个图形 s 中
- 针对一个图形类的每个属性 **x** ：
  - **x()** 提供该属性的当前值
  - **set_x()** 设置该属性为一个给定的新值
  - e.g.,

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

逻辑上相同的操作/函数采用相同的名称！

```
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
```

- 我们为什么不使用win.add(ln)?
  - add() 复制信息; attach() 仅仅是创建一个引用
  - 我们不能够在 attach 一个对象后修改它，但 add 操作则可以



一致性的访问保护

- 成员数据应该是 **private** 的
  - 数据隐藏 – 不会不小心修改数据
  - 采用 **private** 保护数据，则数据具有一对访问和设置操作

```
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!)
```

- 函数可以是 **public** 的或 **private** 的
  - 对接口采用 **Public**
  - 对仅在类内部使用的函数采用 **Private**

“private”给我们带来的优势是什么?

- 在代码发布后我们仍可以修改我们的实现
- 我们没有向接口类的用户暴露 FLTK 的类型
  - 我们能够使用其他库来代替 FLTK，而不影响用户的代码
- 我们能够在访问函数中提供检查功能(前后向检查、调试等)
  - 不过，我们还没有全面地这样做
- 功能性接口更好阅读和使用
  - e.g., **s.add(x)** 比 **s.points.push_back(x)** 要好一些
- 我们能够增强图形的不变式检查
  - 只有color 和 style 能够修改，points的相对位置则不允许
  - **const** 成员函数
- 这种“封装”的价值对不同的应用领域可能是不同的
  - 但它往往是最好的
  - 这是我们的原则之一
    - i.e., 如无充足的理由，采用这种隐藏形式

“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):
Square s1(Point(100,200),200,300);    // width==200 height==300
Square s2(Point(100,200),Point(200,300));   // width==100 height==100

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

一致性和可读性

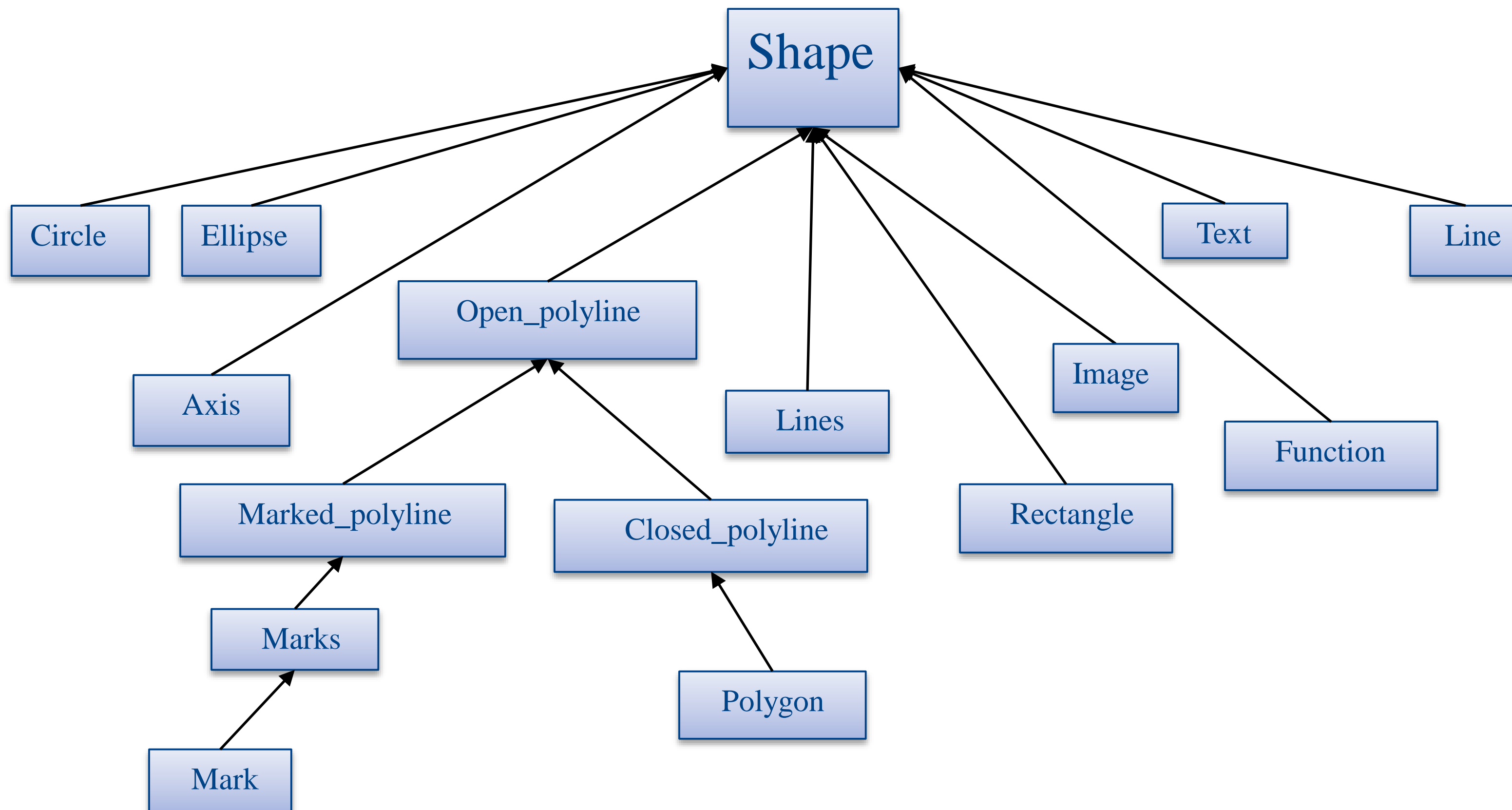
两点一线？直接点

库

- 一些相关类和函数的集合
  - 是其他应用程序的工具程序块
  - 或用于创建更多其他的库(库使用库)
- 好的库会从一个特定的角度直接、清晰地建模其应用领域，强调应用的某些方面
  - 不会试图做好所有的事情(这会导致失败)
  - 针对本图形接口库，我们的目标是简单性、小的图形化数据，以及简单的GUI
- 我们不能孤立地定义库中的每个类和函数
  - 一个好的库会采用一致的风格("规范化")

图形类

- 所有的图形都是“基于”Shape 类的
  - e.g., **Polygon 是一种特殊的 Shape**



类 Shape — 是一个抽象类

❖ 你不能创建一个无意义的 Shape 对象

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

例如：

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

- Protected 表示“只能被本身及其派生类使用”

❖ 代替地，我们将 Shape 用作基类

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

类 Shape

- Shape 将我们的图形对象与“屏幕”连接起来
  - Window 知道 Shapes
  - 所有的图形对象都是一种 Shapes (Window能处理所有的图形对象)
- Shape 是一个类，可以处理所用的颜色和线型
  - 因此，它有 Color 和 Line_style 成员
- Shape 能够存储 Points
- Shape 有一个基本功能，即如何绘制线形
  - 这只需使用它存储的 Points

类 Shape

- Shape 处理 color 和 style
  - 让数据成员是private的，而提供访问接口函数

```
        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;
```

类 Shape

- **Shape 存储 Points**
  - 让数据成员是private的，而提供访问接口函数

```
        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
```

类 Shape

- Shape 自身能够直接访问存储的 points

```
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);
}
```

- 其他类（包括其派生类）只能使用 point() 和 number_of_points()
  - 想一下为什么？

```
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);
}
```

类 Shape (绘图的基本思想)

```
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 …
}
```

共四步，最难和最核心的是第三步

类 Shape (绘图的实现)

```
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
}
```

类 shape

- 在类 Shape 中

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

- 在类 Circle 中

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

- 在类Text 中

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

- Circle, Text, 以及其他类
  - “继承自” Shape
  - 可能“覆盖 (override)” **draw_lines()**

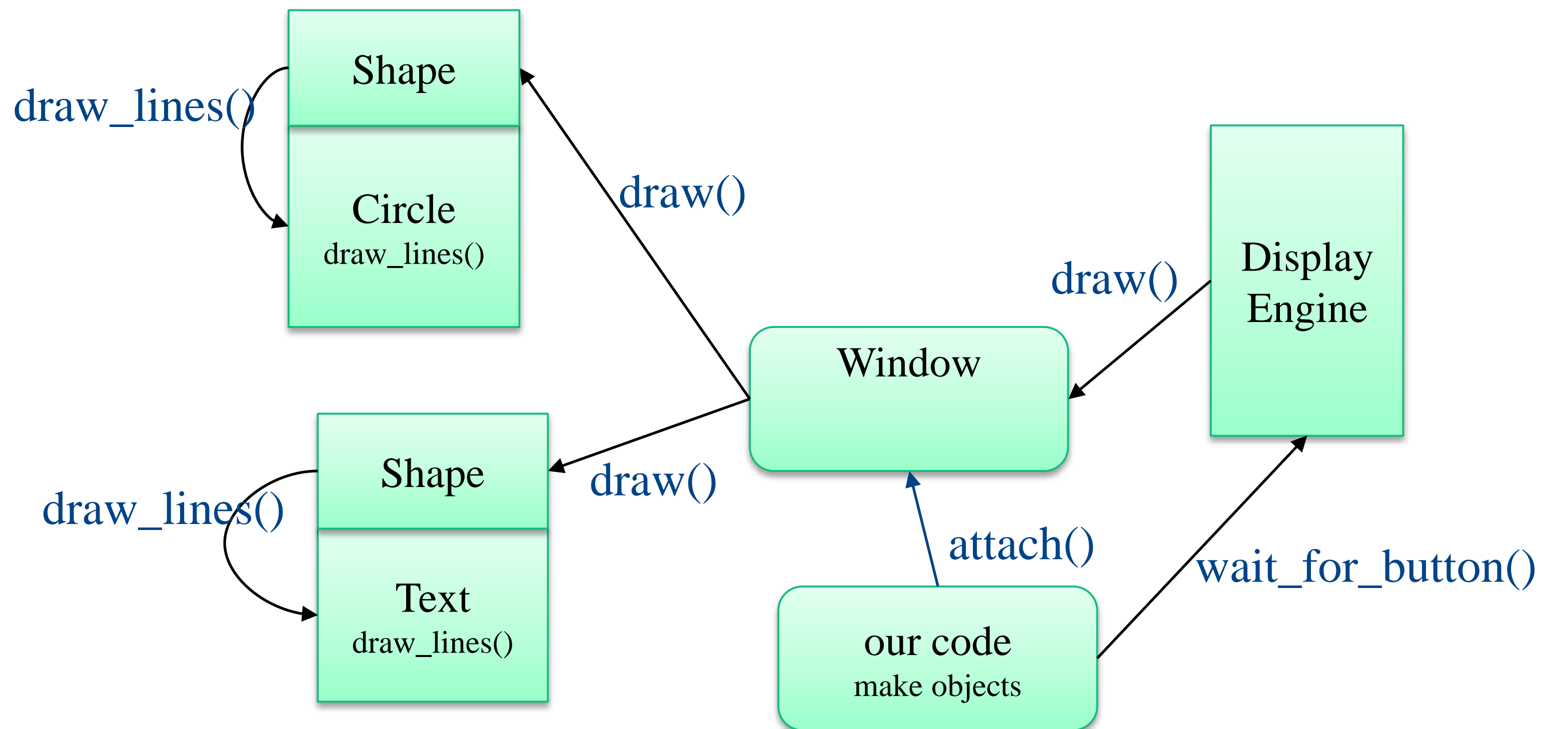
表示派生类中定义一个函数，使之可以通过基类提供的接口进行调用
实现多态，区别于重载 (overload)

类 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 ...
};
```

当设计一个将要作为基类的类时，应禁用它的拷贝构造函数和拷贝赋值操作

完成的显示模型



语言机制

- 面向对象编程 (OOP)的流行定义为:

  OOP == 继承 inheritance + 多态 polymorphism + 封装 encapsulation

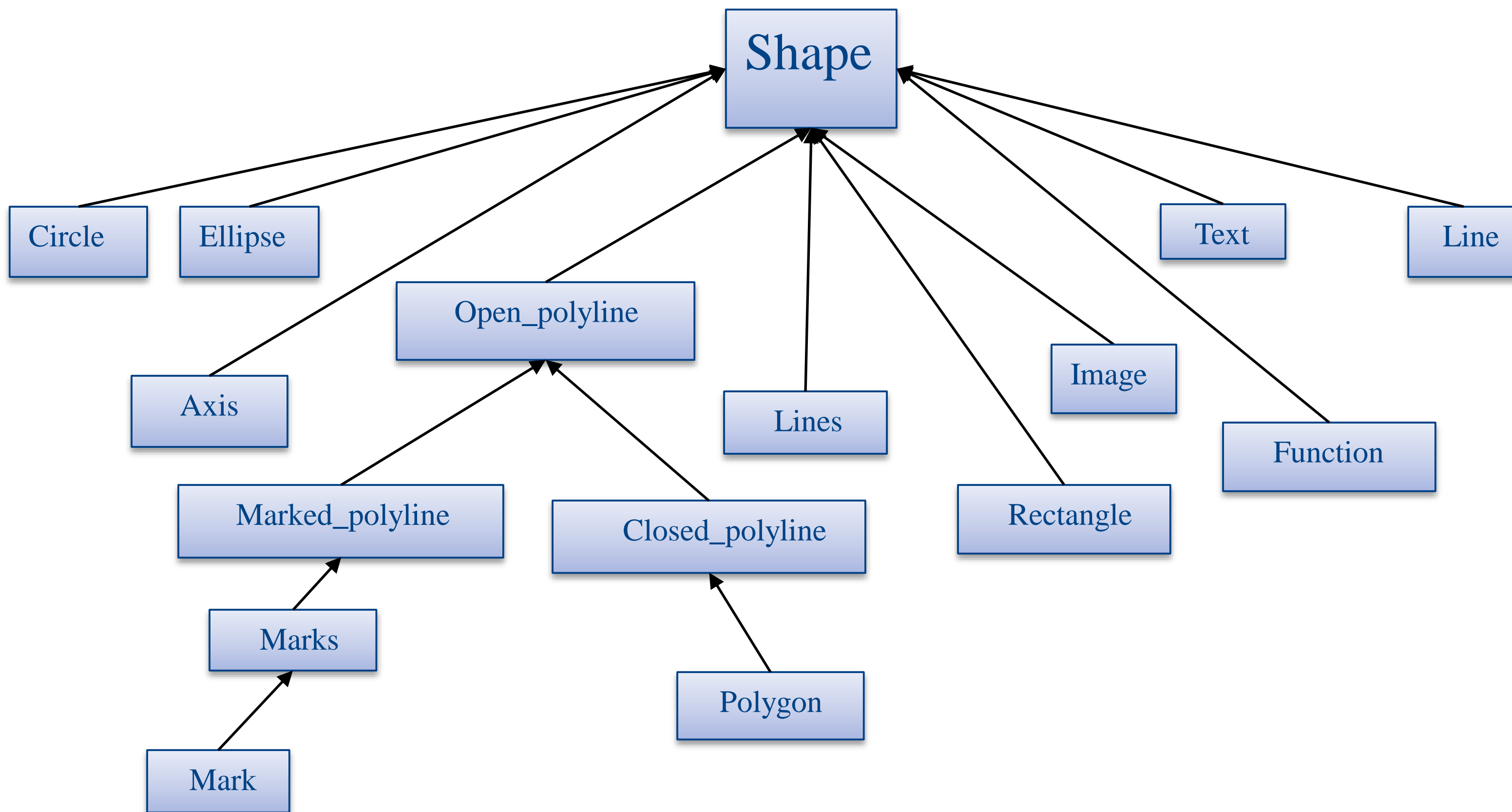
- 基类和派生类 *// inheritance*
  - **struct Circle : Shape { … };**
  - Also called “inheritance”

- 虚函数 *// polymorphism*
  - **virtual void draw_lines() const;**
  - Also called “run-time polymorphism” or “dynamic dispatch”

- Private 和 protected *// encapsulation*
  - **protected: Shape();**
  - **private: vector<Point> points;**

一个简单的类分级

- 我们选择使用一个简单的类分级结构
  - 基于Shape

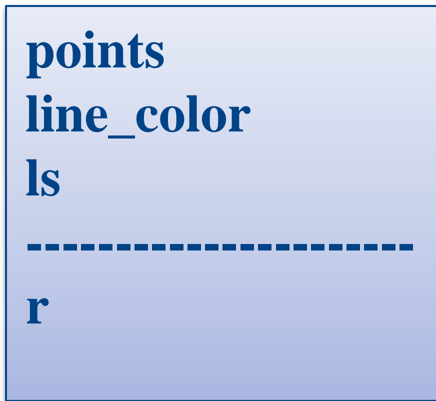


对象布局模型

- 一个类的数据成员是简单地添加到它的基类数据成员之后，e.g., Circle 是一种 Shape，仅增加了半径变量 r

Shape:

Circle:

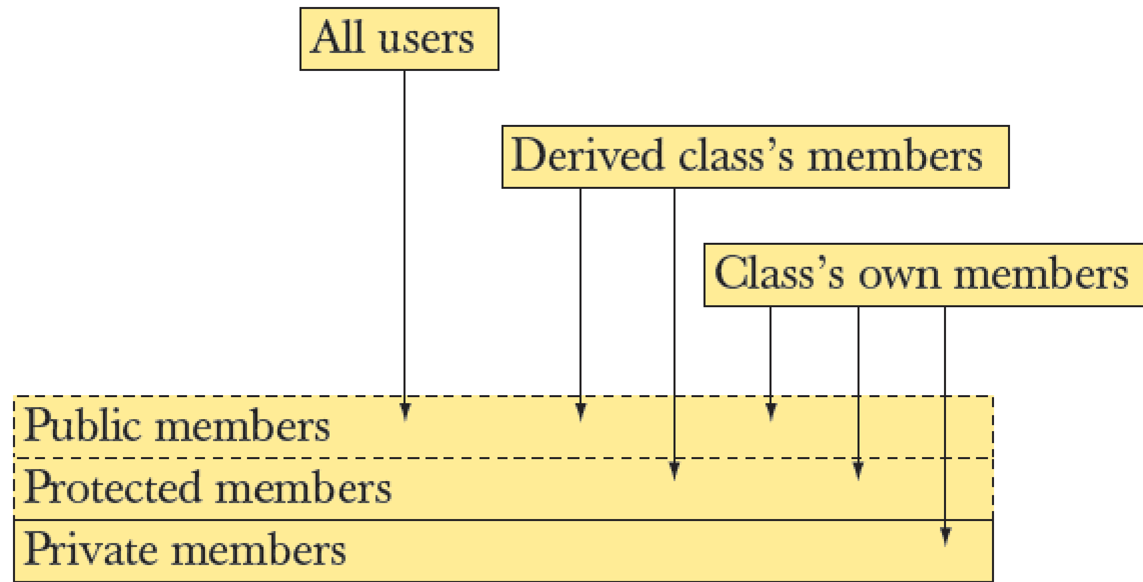


继承的好处

- 接口继承
  - 需要Shape对象参数(通常为Shape&)的函数能够接受其所有派生类的对象参数
  - 简化使用方式
  - 我们能够自由地向程序中添加继承自Shape的新类，而无需重写已有的用户代码
    - 无需修改老代码的增加是程序设计维护的“圣杯”之一
- 实现继承
  - 简化继承类的实现
    - 共用功能只需在一处提供即可
    - 修改时只需在一处改动即可，并能够保持一致
      - 另一个“圣杯”

当你需要拷贝代码时，往往是设计出了问题！

访问模型



- 一个成员（数据、函数或类型成员）或一个基类都可以是
  - Private、protected 或 public 的

纯虚函数

- 有时，接口中的函数无法实现
  - e.g. 需要的数据被隐藏在派生类中
  - 我们必须保证派生类实现这类函数
  - 设置“纯虚函数”的语法：=0
- 这是定义抽象接口（纯接口）的通常方式

```
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
```

纯虚函数

- 通常，一个纯接口出现在基类中
  - 构造和析构函数的技术细节在 chapters 17-19 中再详细说明

```
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
```

拷贝的技术细节

- 如果你不知道如何去拷贝一个对象，就禁止拷贝操作
  - 典型情况下，抽象类是不允许拷贝的

```
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)
}
```

覆盖的技术细节

- 为了覆盖一个虚函数，你需要
  - 一个虚函数
  - 完全相同的名称
  - 完全相同的参数类型

```
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
};
```

Next

- 绘制函数和数据