Week 10 : Class Design, and GUI Classes

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 10

Outline

Graphics class design

Graphing simple functions

Graphical User Interfaces

Outline

Graphics class design

- Introduction to class design
- The `Shape` class
- Base and derived classes
- Benefits of object-oriented programming

Graphing simple functions

Graphical User Interfaces

Design principles[1]

- We will use graphics classes as an example of class design
- Want to have an interface that is simple and consistent to use
- Part of program design is to represent concepts in application directly in code
- We used types:
  - `Window` – a window presented by OS
  - `Line` – a line on the screen
  - `Point` – a coordinate point
  - `Color` – a color on the screen
  - `Shape` – things in common for all shapes

[1] Notes based on: B.Stroustrup, "Programming Principles and Practice Using C++ ," Addison-Wesley (2009) Chapters 14-15.

Abstract types

- `Shape` is a generalization – a purely abstract notion
- We never make a `Shape`, only things that are derived from it (ie. `Line`, `Circle`, `Rectangle`, etc.)
- The set of graphics interface classes is a library
- They are meant to be used together – design of one of classes depends on the others
- Library is not complete – otherwise it would be enormous – instead it is simple and *extensible*
- That means it would be easy to extend its functionality

Setting bounds for library

- We can't hope to create a library that covers all possible graphics
- Things that we add to the library depend on domain we want to apply the classes to
- For example a library for displaying geographic information might need classes for:
  - Showing vegetation
  - Showing provincial and political boundaries
  - Showing roads
  - Showing rivers
  - Showing elevations
- All those might be handy to have, but if you need something for Scientific Visualization, Aircraft control displays, or the like you would need a different set of classes

Small simple classes vs kitchen sink classes

- We provided lots of little classes: `Open_polyline`, `Closed_polyline`, `Polygon`, `Rectangle`, `Marked_polyline`, `Marks`, and `Mark`
- A single class `polyline` with lots of arguments could have represented all of these shapes
- May have even had ways to mutate one of these shapes into another
- Extreme of this would be to only have the `Shape` class that takes lots of arguments and represents any of the shapes we defined
- Single class providing everything would leave it to user to represent the shapes on the screen
- Having many smaller classes, we feel, makes it simpler to understand

Class Operations

- Minimal set of operations is provided
- Ideal is minimal interface to allow us to draw things
- Additional operations provided by non-member functions or additional classes
- Want to have common style for operations
- Logically identical operations have same name, for example every function that adds points, lines to a shape is called `add()`
- Every function that draws lines is called `draw_lines()`
- Such uniformity simplifies things by having fewer details to remember
- Sometimes such uniformity may even allow us to write code that works for many different types (called *generic* code)

Class Operations continued

- An example of this is use of `Point` to specify location of a `Shape`, ie. in constructors:

```
1 Line aa{ Point{100,200}, Point{300,400} };
2 Mark mm{ Point{100,200}, 'x' };
3 Circle cc{ Point{200, 200}, 250 };
```

- We could have provided several different functions:

```
void draw_line( Point p1, Point p2 );
void draw_line( int x1, int y1, int x2, int y2 );
```

- We have consistently used first style above for improved type checking

Class Operations continued

- Also, use of `Point` for location of shape saves us from confusion over whether a coordinate or a size of the shape is the required
- For example function to draw a rectangle:

```
1 // our style
2 draw_rectangle( Point{100,200}, 300, 400 );
3 // alternative style
4 draw_rectangle( 100, 200, 300, 400 );
```

- Our style takes a point, width and height
- Other style could be defined by two points (100,200), (300,400), or a point (100,200) followed by width and height
- Using `Point` helps avoid this confusion
- Also for consistency:
  - ask for coordinates in order x then y
  - ask for width before height

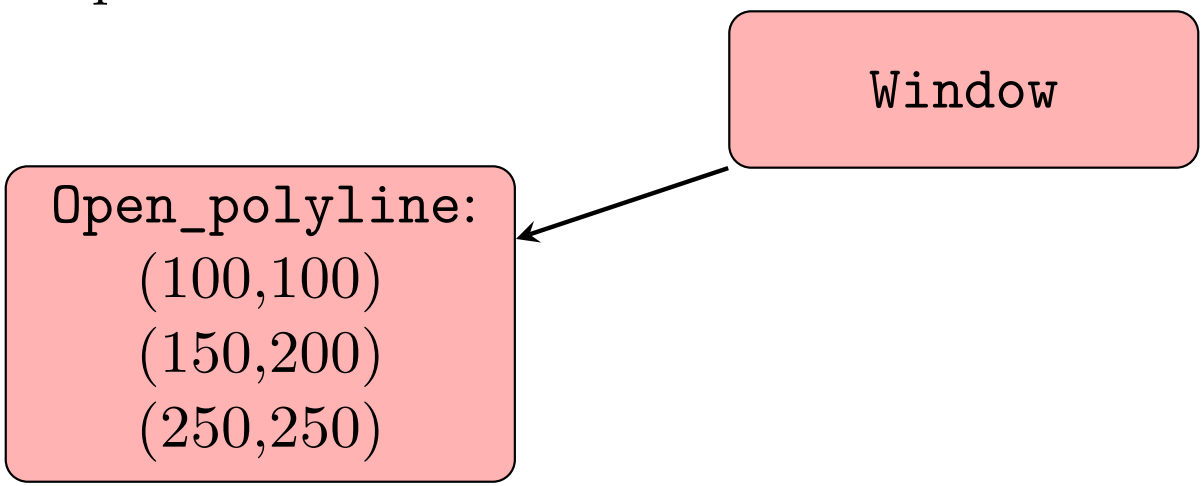
Naming of operations

- Operations that do different types of things should have different names
- Seems obvious... but why do we use `attach` to add a `Shape` to a window, and `add` to add points to a `Shape`?
- The difference is as follows:
  - We rarely keep the points we let the shape take care of them – we don't change the point after we have added it to the shape
  - Shapes `attached` to a window however can be changed by us after we "attach" them to the window
- For example:

```
1 Open_polyline opl;
2 opl.add( Point{100,100} );
3 opl.add( Point{150,200} );
4 opl.add( Point{250,250} );
5 win.attach(opl);
```

Naming continued

- We created a connection between our window `win` and the shape `opl`
- `win` does not make a copy of `opl` so we must keep that shape until we `detach` it from the window



- `add` uses pass by value to copy the point into the shape
- `attach` uses pass by reference to share a single object

Implications for pass by reference

- We can't create objects, attach them to the window, and have them go out of scope:

```
1  void f ( Simple_window & w ){
2    Rectangle r{ Point{100,200},50,30};
3    w.attach(r);
4  } // oops r has gone out of scope
5  int main(){
6    Simple_window
         win{Point{100,100},600,400,"Mywin"};
7    // ...
8    f( win ); // going to get in trouble!
9    // ... added r, but now it doesn't exist
10   win.wait_for_button();
11   return 0;
12 }
```

Mutability

- Who can modify the data in our class and how?
- Only let the class itself change its state
- The `public/private` let us identify whether only the class itself or a user of the class can change a particular value in the class
- Another tag called `protected` can be added to identify parts of the class that can be accessed and changed by any classes inherited from this one
- On top of defining the data that is in the class, we must decide if it can be modified after the object is created

Mutability example

- For example the `Circle` class does not allow the radius to be changed after creation:

```
1 struct Circle {
2   // ...
3   private:
4     int r; //radius
5 };
6 //...
7 Circle c{ Point{100,200}, 50 };
8 c.r=-9; //Error: Circle::r is private
```

- By not allowing value to change we can make sure ridiculous values aren't supplied
- Note that not all of the types defined in the interface library are so careful – that keeps the code a bit shorter
- Only enough to make sure there is no data corruption
- Also `Window` objects are simply an output device – we add our shapes to it, but never query it for things on the window

The `Shape` class

- `Shape` class represents a shape on the `Window` and handles:
  - Connection to `Window` which provides the connection to OS and physical screen
  - Deals with color and style of lines and fill (holds `Line_style`, `Color` for lines, `Color` for fill
  - Holds sequence of `Points` and basic drawing of lines between them
- Maybe it is already doing too much... but it is simple enough for our purposes

The Shape class interface public part

```
1  // Shape deals with color and style,
2  // and holds sequence of lines
3  class Shape  {
4    public:
5      void draw() const; //deal with color and draw_lines
6      // move the shape +=dx and +=dy
7      virtual void move(int dx, int dy);
8      void set_color(Color col) { lcolor = col; }
9      Color color() const { return lcolor; }
10     void set_style(Line_style sty) { ls = sty; }
11     Line_style style() const { return ls; }
12     void set_fill_color(Color col) { fcolor = col; }
13     Color fill_color() const { return fcolor; }
14     Point point(int i) const { return points[i]; }
15     int number_of_points() const { return
           int(points.size()); }
16     // Prevent copying
17     Shape(const Shape&) = delete;
18     Shape& operator=(const Shape&) = delete;
19     virtual ~Shape() { }
20     //...
```

The Shape class private and protected parts

```
class Shape {
  //...
  protected:
    Shape() { }
    // add() the Points to this Shape
    Shape(initializer_list<Point> lst);
    void add(Point p){ points.push_back(p); }
    void set_point(int i, Point p) { points[i] = p; }
    // simply draw the appropriate lines
    virtual void draw_lines() const;
  private:
    // points not used by all shapes
    vector<Point> points;
    Color lcolor { fl_color() };
    Line_style ls {0};
    Color fcolor {Color::invisible};
};
```

An abstract class

- For a fairly complex class, it still only has four data members and 15 member functions
- Consider the constructors first:

```
protected:
  Shape(){ }
  // add() the Points to the Shape
  Shape( initializer_list< Point > lst );
```

- Constructors are `protected` – they can only be used directly from classes derived from `Shape` (using the `: Shape` notation)
- `Shape` can only be used as a base for classes – by making the constructor `protected` we can't make a `Shape`:

```
      Shape ss;
```

- A class is *abstract* if it can only be used as a base class

Notes on `Shape` class

- Notion of a shape is an abstract concept
- Question: What does a shape look like?
- Response: What shape?
- Default constructor of `Shape` sets members to default values and vector of `Points` starts out empty
- Initializer-list constructor uses defaults, and then adds to vector of points:

```
1 Shape::Shape( initializer_list<Point> lst ){
2   for ( Point p : lst ) add( p );
3 }
```

Notes on `Shape` class

- The declaration:

```
1 virtual ~Shape(){ }
```

- defines a virtual destructor
- a destructor is called when an object of the class type is destroyed (by going out of scope or by being `delete`ed)
- We will discuss what the `virtual` keyword means in a few slides

Access control

- All of the data members of `Shape` are `private`:

```
vector<Point> points;
Color lcolor { fl_color() };
Line_style ls {0};
Color fcolor {Color::invisible};
```

- Initializers don't depend on constructor arguments so they are given default values
- If we want to provide users of the class a way to access this private data, provide "set" and "get" methods, for example:

```
void Shape::set_color( Color col ){
  lcolor = col;
}
Color Shape::color() const {
  return lcolor;
}
```

- An alternate convention might have been `SetColor`, and `GetColor`
- In any case, choose convenient names for these functions since they are part of the public interface

Access control continued

- `Shape` keeps a vector of `Points` called `points`
- Function `add()` is used to add Points to the vector:

```
1 void Shape::add( Point p ) {
2   points.push_back( p );
3 }
```

- Note that derived classes don't have direct access to `points`
- `add()` was made `protected:` to allow derived classes to add to the points
- That way all the derived classes have a way to add to the points
- Keeps users from directly adding points
- Wouldn't want user to add points to `Circle` and `Rectangle`
- `Lines` only allows users to add pairs of `Points`

Access control continued

- `set_point()` is also `protected` – only derived class knows what point means and whether it can be changed
- `point()` and `number_of_points()` were made public since looking at the values seemed harmless

```
1 void set_point(int i, Point p) {
2   points[i] = p;
3 }
```

- Since it is a protected member assume we will be careful in calling `set_point()` with index within the vector length
- In derived class member functions use `point()` function for example in `Lines::draw_lines()`:

```
1 void Lines::draw_lines() const {
2   for (int i=1; i<number_of_points(); i+=2)
3     fl_line(point(i-1).x,point(i-1).y,point(i).x,point(i)
4 }
```

Code efficiency

- Does adding all these simple access functions make our code less efficient to run?
- **No.** The compiler will "inline" them, meaning that for example `number_of_points()` takes same amount of memory and time as calling `points.size()` directly.

Why use access control?

- We could have provided a close to minimal version of `Shape`:

```
struct Shape {
  Shape();
  Shape( initializer_list< Point > );
  void draw() const;
  virtual void draw_lines() const;
  virtual void move( int dx, int dy );
  virtual ~Shape();
  vector<Point> points;
  Color lcolor;
  Line_style ls;
  Color fcolor;
};
```

- What do we gain by adding `private:`, `public:`, and 12 member functions?

Why use access control?

- Ensures `Shape` doesn't change in unanticipated ways
- Also allows updating data members without the user of the class having to change any code
- For example, suppose an early version of `Shape` used:

```
1 Fl_Color lcolor;
2 int line_style;
```

- For one it exposes user of class to part of `fltk` library by having the `fltk` type `Fl_color`
- If we now change `Fl_Color` to `line_style` to accommodate colour and width of the line, user will have to update code to use the new types
- Another advantage is notation convenience, eg:

```
1 s.points.push_back(p);   //becomes:
2 s.add(p);
```

Drawing shapes

- `Shape`'s most basic job is to draw shapes:

```
1 void draw() const;
2 virtual void draw_lines() const;
```

- Other functionality of `Shape` could be removed, but drawing the shape is essential concept of shape.
- From users perspective the two functions are:
  - `draw()` applies the style and color and calls `draw_lines()`
  - `draw_lines()` puts the pixels on the screen

Shape::draw()

```
1 void Shape::draw() const {
2   Fl_Color oldc = fl_color();
3   // there is no good portable way of retrieving the
        current style
4   fl_color(lcolor.as_int());
5   fl_line_style(ls.style(),ls.width());
6   draw_lines();
7   fl_color(oldc);
8   fl_line_style(0);
9 }
```

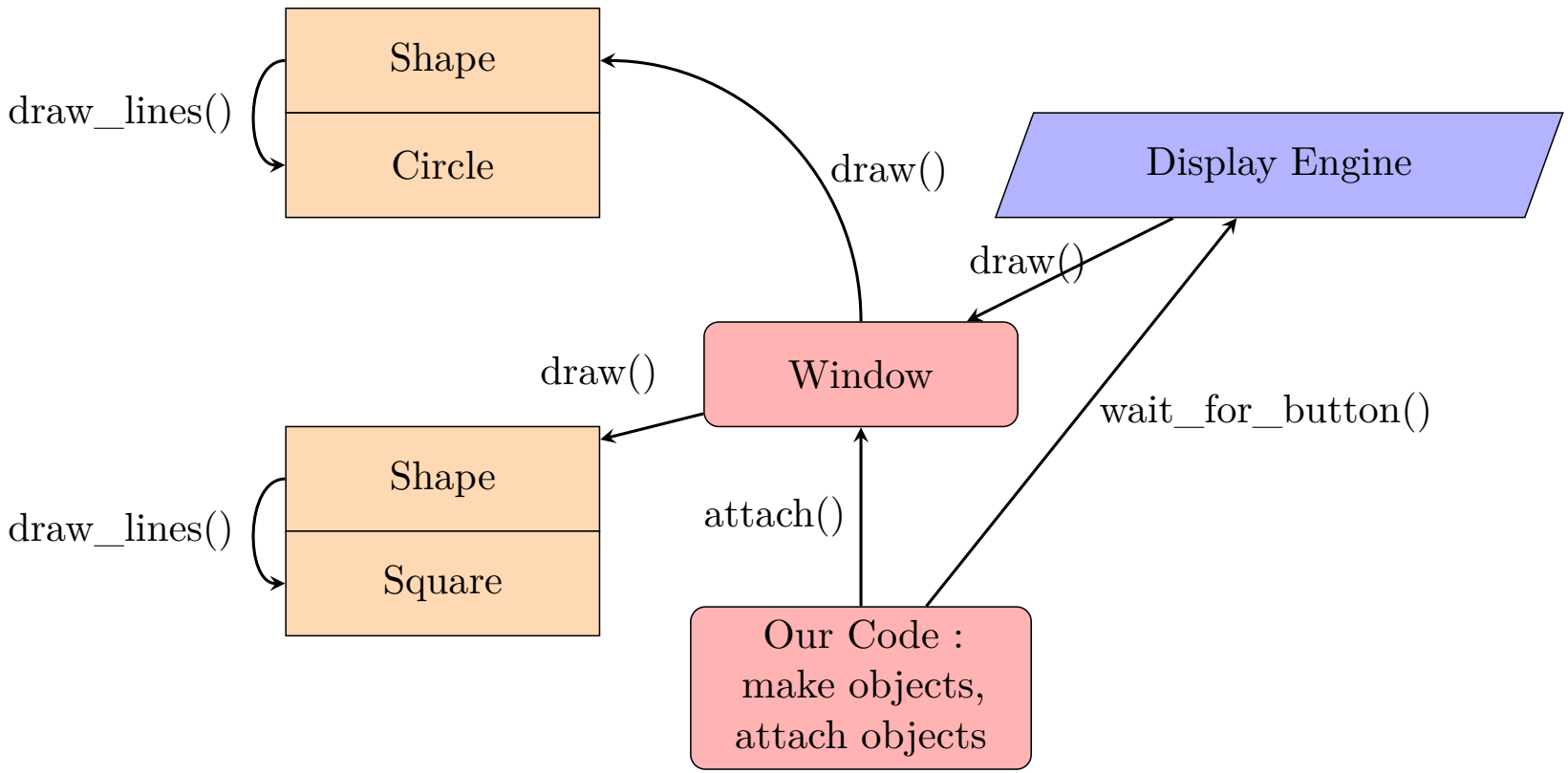
- `fltk` had no way to get current line style, so used `fl_line_style(0)` to reset to default at end
- `Shape::draw` does not change fill color or visibility – that is done by individual `draw_lines`

Shape::draw_lines()

- Realize that **Shape** won't be able to draw all of the shapes – it doesn't know the details of all the derived shapes
- Instead each derived type **Text**, **Circle**, **Rectangle**, etc can provide its own version of **draw_lines()**
- After all the derived types know the details of what they are supposed to represent
- For example a **Circle** defines its own **draw_lines()** and we want to use that instead of **Shape::draw_lines()**
- That is why we defined **virtual Shape::draw_lines()**

```
1 struct Shape { //...
2   // let each derived class define its
3   // own draw_lines() if it chooses:
4   virtual void draw_lines() const;
5 };
6 struct Circle : Shape { //...
7   // override Shape::draw_lines()
8   void draw_lines() const;
9 };
```

Completed model of display



Shape::move()

- Function to move every point stored relative to current position:

```
1 void Shape::move(int dx, int dy) {
2   for (unsigned int i = 0; i<points.size(); ++i) {
3     points[i].x+=dx;
4     points[i].y+=dy;
5   }
6 }
```

- Also declared as `virtual` since derived classes may have data that needs to be moved that `Shape` doesn't know about
- For example `Axis` needs to move its ticks.

Copying and mutability

- Notice that `Shape` declared copy constructor and copy assignment as `deleted`:

```
Shape( const Shape & ) = delete;
Shape& operator=( const Shape & ) = delete;
```

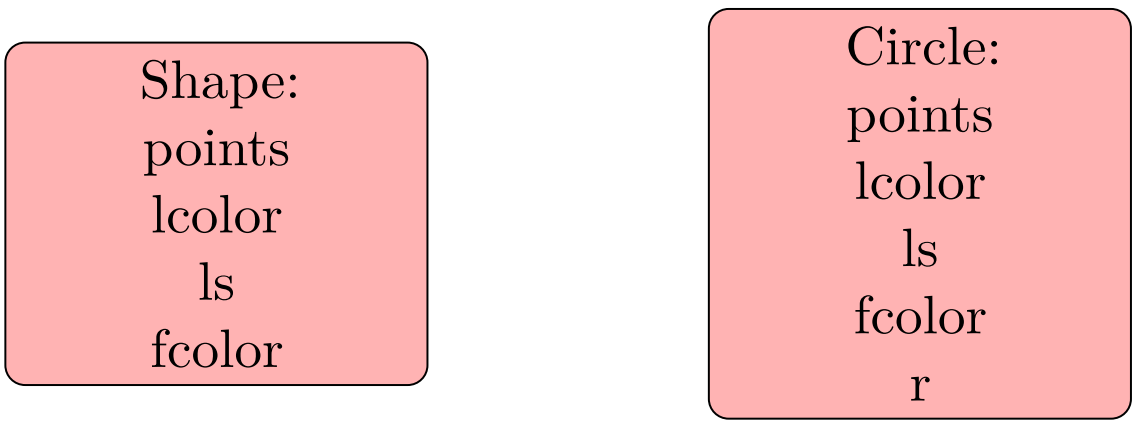
- Eliminates otherwise default copy operations, eg:

```
void f( Open_polyline & op, const Circle & c ){
  Open_polyline op2 = op; //error: shape copy
      deleted
  vector<Shape> v;
  v.push_back( c ); //error: shape copy deleted
  //...
  op = op2; //error: shape assign deleted
}
```

- Why would we delete that functionality?!

Copying and mutability

- Why did we delete default copy? – now we can't push the objects into a vector easily
- Reason is to avoid "trouble"
- ie. in previous example `v.push_back(c)` tried to add `Circle` to vector of `Shapes`
- But `Circle` has radius and `Shape` does not
- Therefore `sizeof( Circle ) > sizeof( Shape )`
- Circle would be "sliced" and future use of shape would likely cause trouble



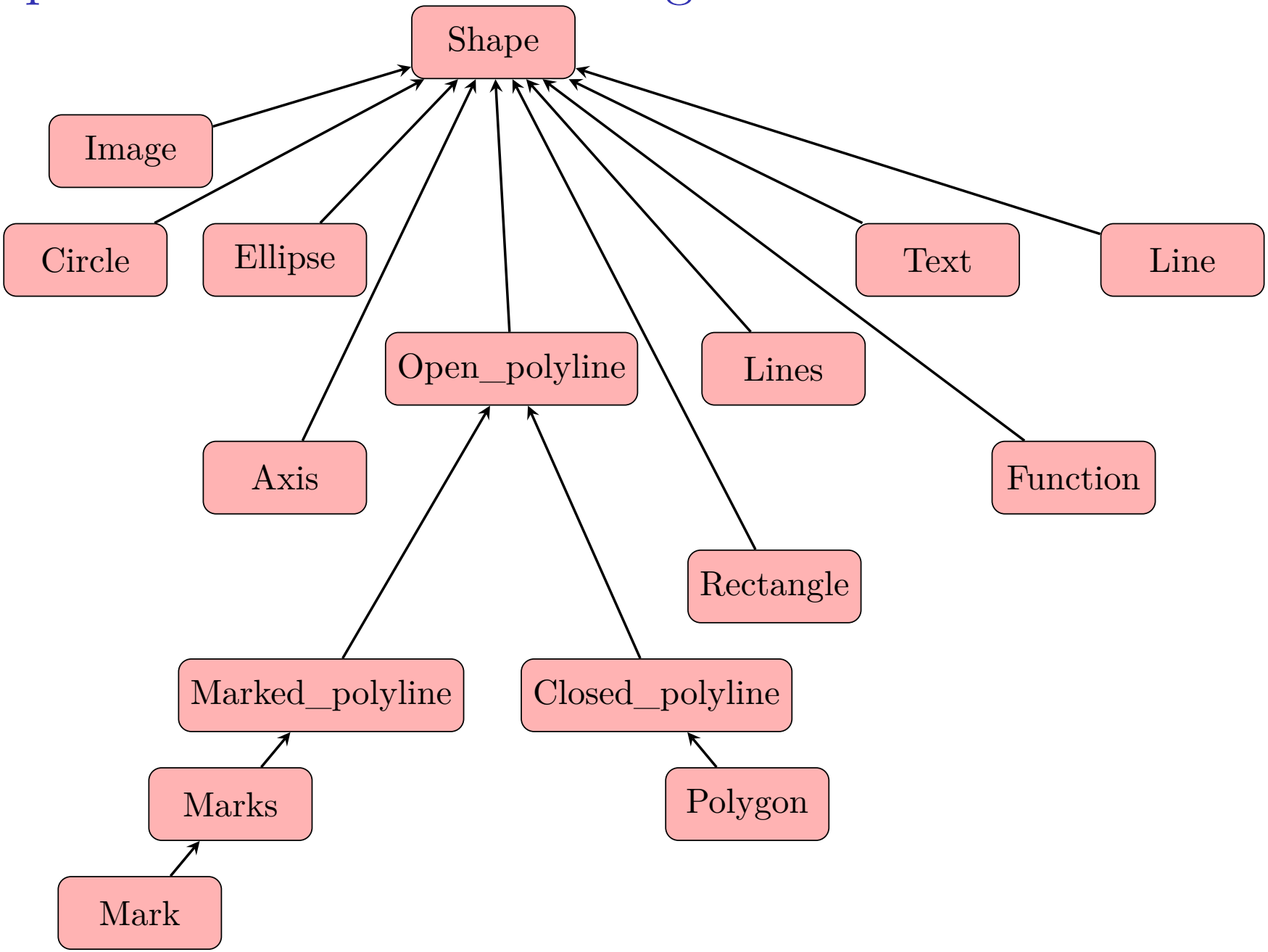
Copying and mutability

- Slicing is technical term for losing some information by default copy of derived class into base class
- If we want to copy objects whose default copy operation is disabled, provide a `clone()` method
- `clone()` method would then need to have read access to all the members of the class

Base and derived classes

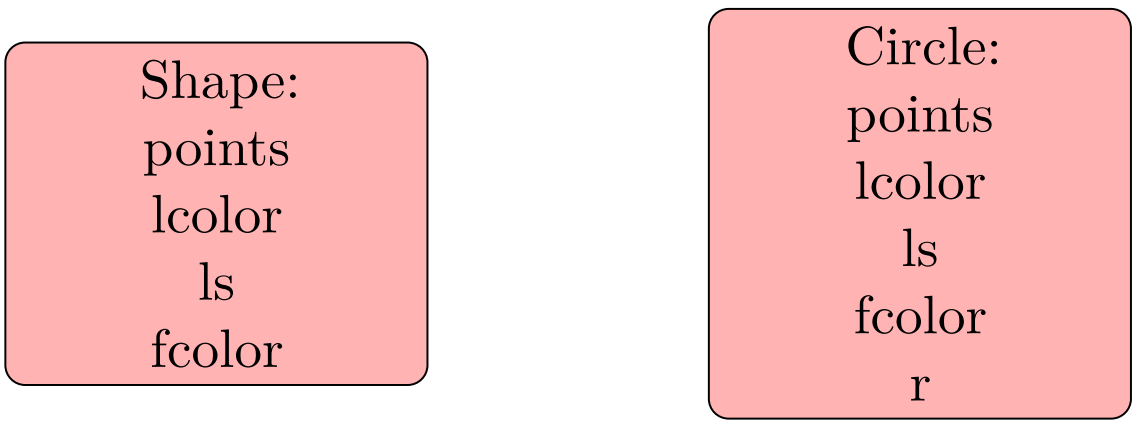
- In defining our graphics interface classes we used three language features:
  - *Derivation*: We build one class from another so that new class can be used in place of original. Ie. a `Circle` can be used in place of a `Shape`
  - *Virtual functions*: Ability to define function in base class and have same named function in derived class called when user calls the base class function.
    This is called *run-time polymorphism*, *dynamic dispatch*, or *run-time dispatch*
  - *Private and protected members*: We kept implementation details private to protect them from direct use that could complicate maintenance – this is often called *encapsulation*
- Use of inheritance, run-time polymorphism, and encapsulation is most common definition of *object-oriented* programming.

Graphics class inheritance diagram



Object layout in memory

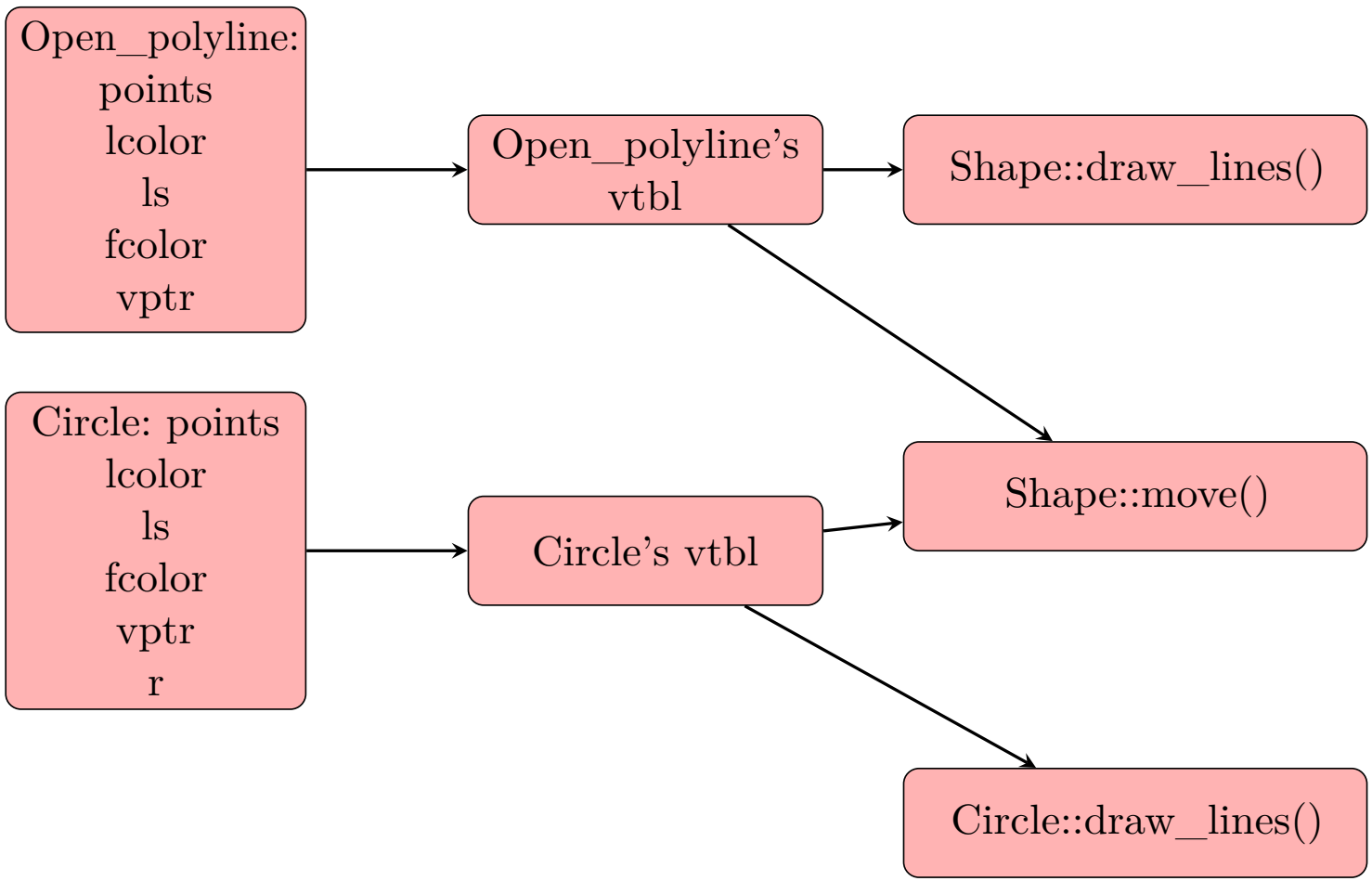
- Data members are stored one after another in memory
- Derived class's added data members get added after those in base class
- For example we saw layout of `Shape` and `Circle`:



- For classes with `virtual` functions, also need to add address of the virtual functions
- That way it knows which function to call
- Table of addresses is called the `vtbl` for "virtual table" or `vptr` for "virtual function table"

Object layout in memory

- Picture for memory of `Circle` and `Open_polyline`:



Growing the `vtbl`

- `draw_lines()` is first virtual function in `Open_polyline` so it gets first slot in `vtbl`
- `move()` is second, so it gets second slot and so on
- Can add as many virtual functions to `class` as you want, and `vtble`
- Cost to call a virtual function is two memory accesses – one to get to vtbl – and one to call the function
- This is simple and fast

Deriving classes and defining virtual functions

- We specify a class derived from another by mentioning a base class after the class name
- For example:

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

- By default the members of `struct` are `public` and that will include `public` members of the base class.
- Above is equivalent to:

```
struct Circle : public Shape { /* ... */ };
```

- Beware of forgetting public when you need it, for example:

```
class Circle : Shape { public: /* ... */ };
```

- Is probably an error, and Shape's public functions are inaccessible to `Circle`

Defining virtual functions

- virtual functions are declared in class declaration
- virtual keyword is not allowed in function implementation:

```
1  struct Shape {
2    //...
3    virtual void draw_lines() const;
4    virtual void move();
5    //...
6  };
7  // *** below is error ***
8  virtual void Shape::draw_lines() const { /*...*/ }
9  // *** below is ok ***
10 void Shape::move{ /*... */ }
```

Overriding

- To override virtual function in base class, need exact name and argument list as base class:

```
struct Circle: Shape {
  // probably mistake (int argument?)
  void draw_lines( int ) const;
  // probably mistake (misspelled name?)
  void drawlines() const;
  // probably mistake (const missing?)
  void draw_lines();
  //...
}
```

- Above functions would be seen as three different functions than the virtual function declared in `Shape`

Short example of overriding

```
1  struct B{
2    virtual void f() const { cout << "B::f "; }
3    void g() const { cout << "B::g "; }
4  };
5  struct D:B{
6    // below overrides B::f
7    void f() const { cout << "D::f "; }
8    void g(){ cout << "D::g "; }
9  };
10 struct DD:D{
11   // below doesn't override D::f
12   // since it is not const
13   void f() { cout << "DD::f "; }
14   void g() const { cout << "DD::g "; }
15 };
```

Short example of overriding continuing

```
1  void printfg( const B& b ){
2    // D and DD derive from B, so can pass
3    // either of those off as a B
4    b.f();   b.g();  cout<<"  ";
5  }
6  int main(){
7    B b;  D d;   DD dd;
8    printfg( b );
9    printfg( d );
10   printfg( dd ); cout<<endl;
11   b.f();    b.g(); cout<<"  ";
12   d.f();    d.g(); cout<<"  ";
13   dd.f();  dd.g(); cout<<endl;
14 }
```

- Result is:

```
B::f B::g   D::f B::g   D::f B::g
B::f B::g   D::f D::g   DD::f DD::g
```

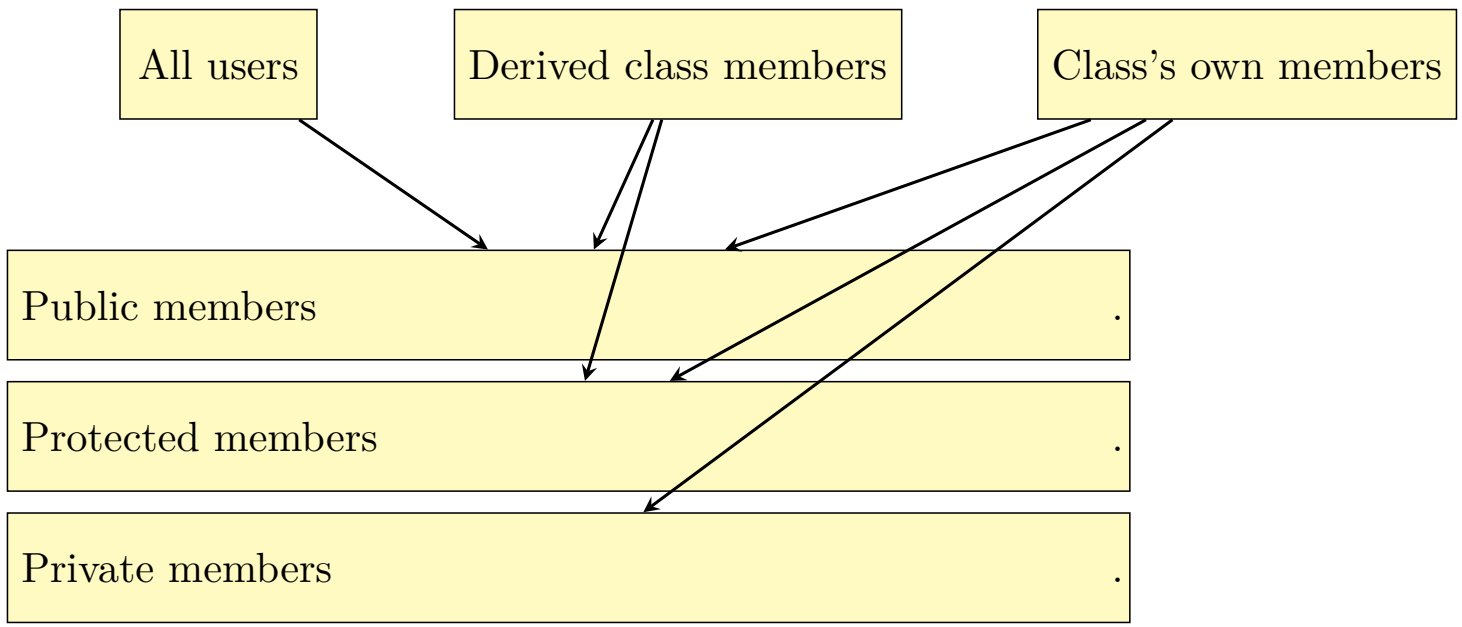
Overriding continuing

- We can ask compiler to check that a function we want to have override the base class one does using `override`:

```
1  struct B{
2    virtual void f() const { cout << "B::f "; }
3    void g() const { cout << "B::g "; }
4  };
5  struct D:B{
6    // below overrides B::f
7    void f() const override { cout << "D::f "; }
8    // below error: no virtual B::g to override
9    void g() override { cout << "D::g "; }
10 };
11 struct DD:D{
12   // error : below doesn't override D::f
13   // since it is not const
14   void f() override { cout << "DD::f "; }
15   // error : below no virtual DD::g to override
16   void g() const override { cout << "DD::g "; }
17 };
```

Access

- C++ has a simple model for access to members of a class:
  - `private:` its name can only be used by members of the class in which it was declared
  - `protected:` its name can only be used by members of the class it is declared and members of classes derived from it
  - `public:` its name can be used by all functions



Pure virtual functions

- An abstract class is one that can only be used as a base class
- Examples of "abstract" types are: animal, device driver, publication, shape, etc.
- To make a class "abstract" we make its constructor `protected`
- A more common way is to state one or more of its virtual functions *needs* to be overridden:

```
1 class B{
2   public:
3     // pure virtual functions:
4     virtual void f()=0;
5     virtual void g()=0;
6 };
7 B b; // error: B is abstract
```

- The notation of setting the member function `=0` says that the function is "pure" virtual

Pure virtual functions continued

- Since our class `B` has pure virtual functions, we cannot create object of type `B`
- Overriding the pure virtual functions solves this:

```
class D1 : public B {
  public:
    // override virtual functions:
    void f() override;
    void g() override;
};
D1 d1; // Ok
class D2 : public B{
  public:
    void f() override;
    // no g() ?
};
D2 d2;// error: D2 is (still) abstract
class D3 : public D2{
  public:
    void g() override;
};
D3 d3; // ok
```

Benefits of object-oriented programming

- `Circle` is derived from `Shape` – `Circle` is a kind of `Shape`, means either or both of:
  - *Interface inheritance*: A function expecting a `Shape` can accept a `Circle`
  - *Implementation inheritance*: Definition of `Circle` takes advantage of data and member functions offered by `Shape`
- A design that doesn't have interface inheritance is poor
- Interface inheritance's benefits come from code using the interface provided by a base class
- Implementation inheritance's benefits come from the simplification in the implementation of the derived class

Benefits of object-oriented programming

- Note that graphics interface depended on *interface inheritance*:
- Graphics engine calls `Shape::draw()`
- `Shape::draw()` calls `Shape`'s virtual function `draw_lines()`
- Neither the graphics engine, nor the `Shape` knows which kinds of shapes exist
- We can add new `Shapes` to a program without modifying existing code
- This is "holy grail" of code design and maintenance – being able to extend the system without modifying existing system
- *implementation inheritance* was used by adding some useful services in `Shape` class
- However adding too much *implementation inheritance* can make it hard to later update the `Shape` interface

Outline

Graphics class design

Graphing simple functions

Introduction to graphing simple functions

The `Function` class

Graphical User Interfaces

Introduction to functions and graphing

- Our graphing software will be fairly basic, compared to a professional program
- The design techniques, mathematical tools and programming used here will be of longer term use than the actual graphics facilities

Simple functions

- Simplest possible function – a horizontal line at 1 is just a function that returns 1 regardless of what x value we pass it:

```
double one( double x){ return 1; }
```

- Above function is defined by $(x, y)==(x, 1)$
- A line with a slope of $1/2$ might look like:

```
double slope( double x){ return x/2; }
```

- Above function is defined by $(x, y)==(x, x/2)$
- A square function (parabola, with lowest point at $(0, 0)$ is:

```
double square( double x){ return x*x; }
```

- Above function is defined by $(x, y)==(x, x * x)$

Graphing simple functions

- First we define several constants to use in drawing our function:

graphExample.cpp

```
constexpr int xmax=600; //window size
constexpr int ymax=400;
// make center of window (0,0)
constexpr int x_orig=xmax/2;
constexpr int y_orig=ymax/2;
const Point orig{x_orig, y_orig};
// range of function
constexpr int r_min=-10;
constexpr int r_max=11;
// number of points used in range
constexpr int n_points=400;
// scaling factors
constexpr int x_scale=30;
constexpr int y_scale=30;
```

Graphing simple functions, continued

graphExample.cpp

```
1  //... define a bunch of constants first, and
2  //    our C++ functions one, slope, square
3  Simple_window win{ Point{100,100},
4                     xmax,ymax,"Graph Example"};
5  Function fone{ one, r_min, r_max, orig,
6                 n_points, x_scale, y_scale };
7  Function fslope{ slope, r_min, r_max, orig,
8                 n_points, x_scale, y_scale };
9  Function fsquare{ square, r_min, r_max, orig,
10                n_points, x_scale, y_scale };
11 win.attach( fone );
12 win.attach( fslope );
13 win.attach( fsquare );
14 win.wait_for_button();
```

Results of simple graph example

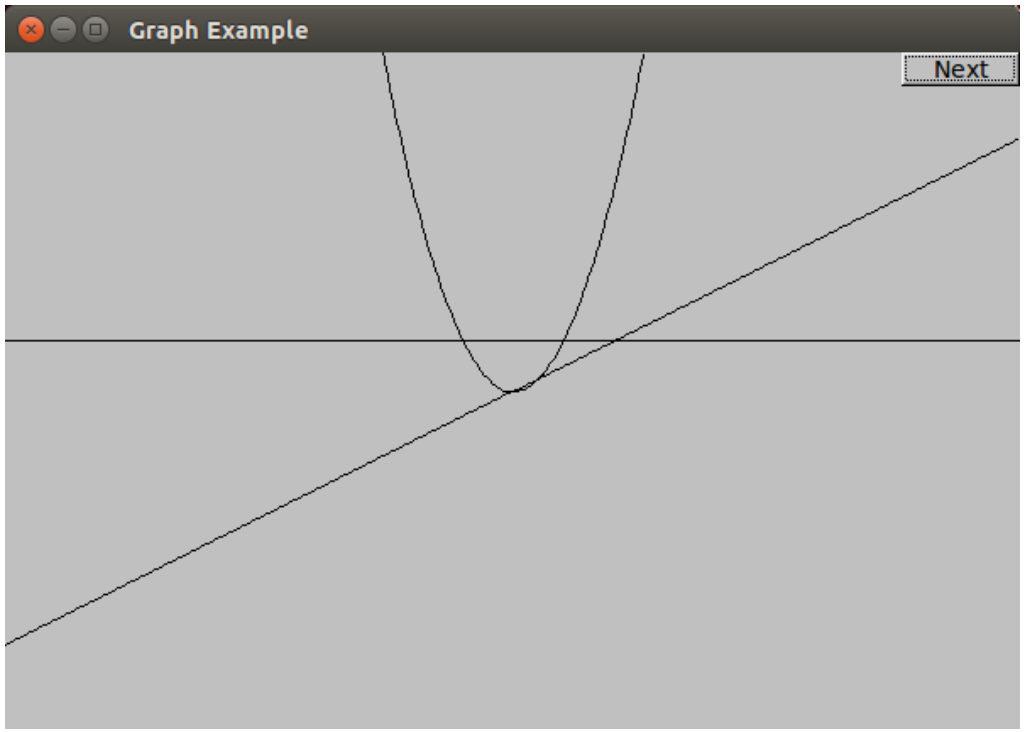


Figure: Result of `testGraph.exe`

Notes on `Function`

- Our use of `Function` requires several arguments:

```
Function fslope{ slope, r_min, r_max, orig,
                 n_points, x_scale, y_scale };
```

- The first argument is a C++ function of one `double` argument that returns a `double` value
- Second and third arguments specify the range of values to use in the function
- The third argument specifies the location on the screen of $(0, 0)$
- The fourth argument specifies how many values to draw within the specified range
- The sixth and seventh arguments specify the scale factor to multiply the x and y values bye when drawing them on the screen
- It is a long list of arguments, but it is needed in this case

Adding labels to identify the functions

- Lets add `Text` to label the functions:

```
1 Text tone{ Point{100, y_orig-40}, "one" };
2 Text tslope{ Point{100, y_orig+y_orig/2-20}, "x/2"
      };
3 Text tsquare{ Point{x_orig-100, 20}, "x*x" };
```

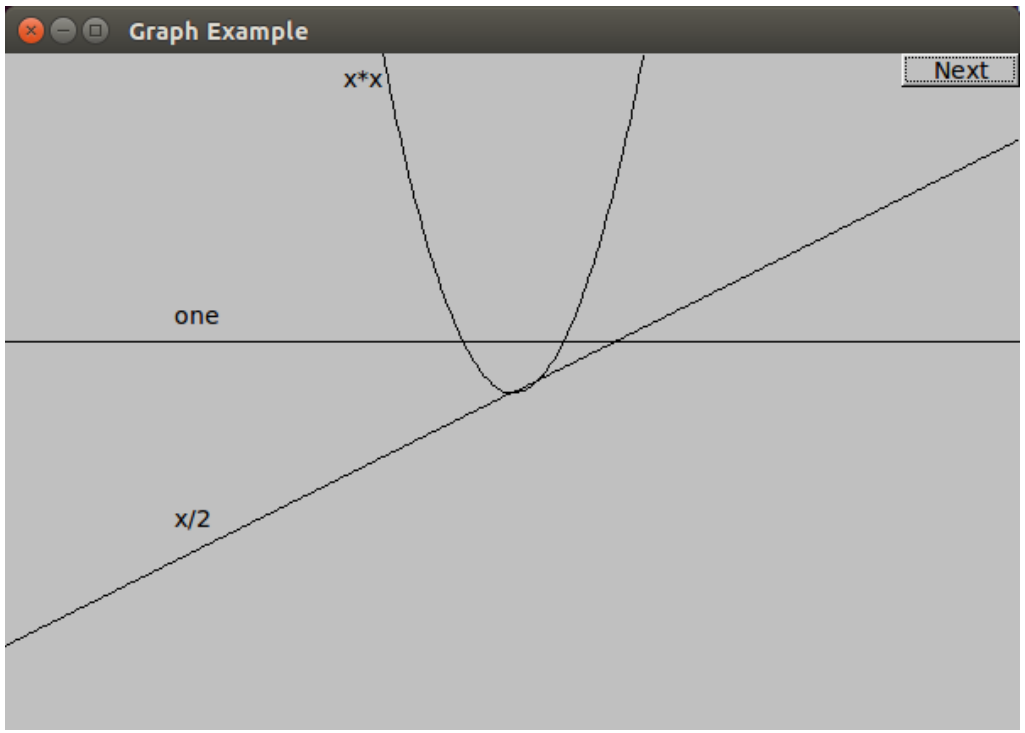


Figure: Updated result of `testGraph.exe`

Adding `Axis` to the window

- To help people figure out the scale for the functions, we need to add axes, for example:

```
1 // make axes slightly shorter than size of window
2 constexpr int xlength = xmax-40;
3 constexpr int ylength = ymax-40;
4 Axis xax{ Axis::x, Point{20,y_orig},
5   xlength, xlength/x_scale,"one notch == 1" };
6 Axis yax{ Axis::y, Point{x_orig, ylength+20},
7   ylength, ylength/y_scale,"one notch == 1" };
```

- The `xlength/x_scale` for number of notches makes each notch represent values 1, 2, 3, etc.
- Here we have put the axes crossing at $(0, 0)$
- May have been better to put axes on edges of plot as is more conventionally done

Result of graph after adding axes

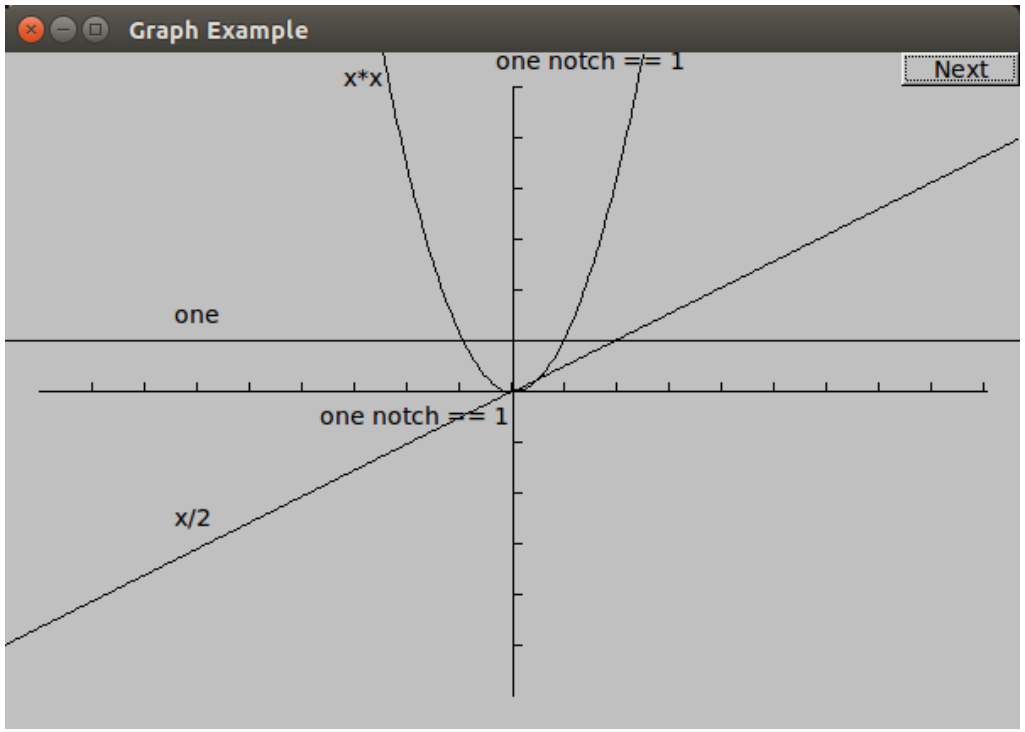


Figure: Updated result of `testGraph.exe`

Adding `Function` interface

- The interface for `Function` is:

```
typedef double Fct(double);
struct Function : Shape {
  // the function parameters are not stored
  Function(Fct f, double r1, double r2,
           Point orig, int count = 100,
           double xscale = 25, double yscale = 25);
};
```

- The `Function` calculates points for the number of points requested and stores them in the Shape
- `xscale` and `yscale` scale the x-coords and y-coords respectively
- Often need to scale values to make them fit window
- Note that `Function` doesn't store any of the values given to it, so we can't redraw it with a different scaling or query it about the function
- `Fct` is a function that takes a double and returns a double

The `Function` constructor

```
1  // graph f(x) for x in [r1:r2). Use count line
2  // segments, with (0,0) at xy. Scale x-coords
3  // by xscale and y-coords by yscale
4  Function::Function(Fct f, double r1, double r2,
5                     Point xy, int count,
6                     double xscale, double yscale) {
7    if (r2-r1<=0) error("bad graphing range");
8    if (count<=0) error("non-positive graphing count");
9    double dist = (r2-r1)/count;
10   double r = r1;
11   for (int i = 0; i<count; ++i) {
12     add(Point( xy.x + int(r*xscale),
13                xy.y - int( f(r)*yscale ) ));
14     r += dist;
15   }
16 }
```

Default arguments in `Function` constructor

- `count`, `xscale`, and `yscale` were given default values
- If the default argument suits our function, then we don't need to supply that value, for example:

```
1 Function s1{   one, r_min, r_max, orig, n_points,
      x_scale, y_scale};
2 Function s2{ slope, r_min, r_max, orig, n_points,
      x_scale };
3 Function s3{square, r_min, r_max, orig, n_points };
4 Function s4{  sqrt, r_min, r_max, orig };
```

- Are equivalent to:

```
1 Function s1{   one, r_min, r_max, orig, n_points,
      x_scale, y_scale};
2 Function s2{ slope, r_min, r_max, orig, n_points,
      x_scale, 25 };
3 Function s3{square, r_min, r_max, orig, n_points,
      25, 25 };
4 Function s4{  sqrt, r_min, r_max, orig, 100, 25,
      25 };
```

More on default arguments

- If a default argument is specified, all following arguments must specify a default argument
- For example this would be an error:

```
1 struct Function : Shape {
2   Function(Fct f, double r1, double r2,
3            Point orig, int count = 100,
4            double xscale, double yscale); // error
5 };
```

- Default arguments don't need to be provided
- Here values were chosen after some time using the Function, and found typically used values

More examples

```
1  double sloping_cos( double x ){ return
       cos(x)+slope(x); }
2  Function s4{ sloping_cos, r_min, r_max, orig, 400, 30,
       30 };
3  s4.set_color(Color::blue);
4  xax.label.move(-160,0);
5  xax.notches.set_color(Color::dark_red);
6  Function f1{log, .0000001,r_max, orig, 200, 30, 30 };
7  f1.set_color( Color::magenta );
8  Function f2{sin, r_min, r_max, orig, 200, 30, 30 };
9  f2.set_color( Color::cyan );
10 Function f3{cos, r_min, r_max, orig, 200, 30, 30};
11 f3.set_color( Color::red );
12 Function f4{exp, r_min, 2., orig, 200, 30, 30};
13 f4.set_color( Color::green );
```

Result of adding many functions to our graph

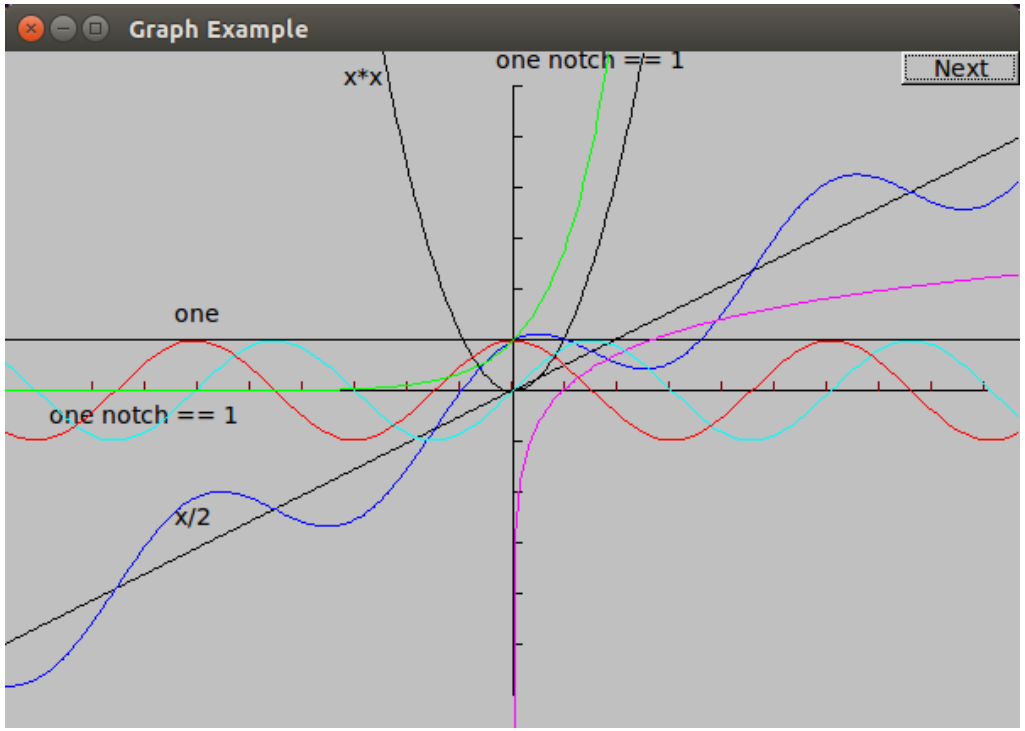


Figure: Updated result of `graphExample.exe`

- Have a look at `graphExample.cpp` and try running it!

Outline

Graphics class design

Graphing simple functions

Graphical User Interfaces
- Introduction to User Interfaces
- `Button` and other `Widgets`
- Control inversion
- Adding a menu
- Debugging GUI code

User interfaces

- We will consider common case of interface to a computer with a keyboard, mouse and screen
- Three main choices for interface:
  - *Console input and output*: We have used this via <iostream> using `cin` and `cout`. It is probably the most commonly used method when solving technical problems.
  - *GUI library*: User interaction by manipulating objects on the screen (pointing, clicking, dragging, dropping, hovering, etc.). This is what we will learn this evening.
  - *Web browser interface*: Markup language where communication between program and screen is again textual. Browser is a GUI application that translates some of the text into graphical elements.

The "Next" button

- We have been using the `Simple_window` class that includes a "Next" button – how does that work?
- Each time we call `win.wait_for_button()` we look at objects on screen until we hit button to get output from the next part of the program
- This seems similar to `cout` to the screen, then wait for input via `cin >> var;`
- Of course implementation is quite different – GUI keeps track of mouse actions (clicking etc.)
- When program wants an action it must:
  - Tell GUI what to look for (eg. "Clicked next button")
  - Tell GUI what to do when someone does that
  - Wait until GUI detects action that program is interested in
- GUI does not just return to our program, it responds to clicking buttons, re-sizing windows, etc.
- We need a way to tell GUI what to do when particular button is clicked

A simple window

- We can ask the GUI to call a function when "something interesting" happens, such as clicking a particular button
- These functions are called "callback functions"
- To do that we must:
  - Define a button
  - Get it displayed
  - Define a function for the GUI to call
  - Tell the GUI about that button and that function
  - Wait for the GUI to call our function

Simple_window's button

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h,
                  const string& title );
    // simple event loop call
    void wait_for_button();
  private:
    // the "Next" button
    Button next_button;
    // implementation detail:
    bool button_pushed;
    // callback for next_button
    static void cb_next(Address,
                        Address pw);
    // action to be done when next_button
    //  is pressed:
    void next();
};
```

Notes on `Simple_window`'s button

- We see that `Simple_window` inherits from `Graph_lib::Window`
- Our button is initialized in the constructor:

```
1 Simple_window::Simple_window(Point xy,
2         int w, int h, const string& title) :
3     Window{xy,w,h,title},
4     next_button{ Point{x_max()-70,0},
5                  70, 20, "Next", cb_next},
6     button_pushed{ false }
7 {
8   attach(next_button);
9 }
```

- We see that location, size and title are passed on to `Window` to deal with
- The `next_button` is then initialized with: a location `Point{x_max-70,0}` , a size (70, 20), a label "Next", and a callback function `cb_next`
- Finally the `next_button` is attached to the window

Notes on `Simple_window`'s button

- The `button_pushed` Boolean is a detail – it is used to keep track of whether the button was pressed since the last time `next()` was called
- In fact the only public part of the implementation is the constructor, and the `wait_for_button()`
- The private method `cb_next()` is what we want the GUI to call when the "Next" button is clicked
- We used the `cb_` prefix to indicate that it is meant to be a callback function
- Recall that program runs on top of several layers (Our program -> GUI interface code -> `fltk` -> Operating system -> Device driver)
- A click detected by mouse driver has to cause `cb_next()` to be called
- We give `Simple_window` the address of `cb_next()` which gets passed down through the software layers so that it can get called

Notes on `cb_next()`

- Type required for callback function is chosen so that is usable from the lowest level of programming, including C and assembler
- A callback returns no value, takes two addresses as args
- Declare a C++ member function that obeys those rules as:

```
1 // callback for next next_button
2 static void cb_next( Address, Address pw);
```

- The keyword `static` makes sure `cb_next()` can be called as an ordinary function
- Ie. it is not a C++ member function invoked for a specific object
- It would have been nice to be able to use a member function, but that's not what we get when talking to lower level code
- The `Address` arguments specify that the function takes two addresses of "somethings" in memory
- First Address we don't need so it is not given a name
- Second Address is of the window containing that "Widget"

Notes on `cb_next()`

- We can use Address as follows:

```
1 void Simple_window::cb_next(Address, Address pw)
2 // call Simple_window::next() for the window
3 // located at pw
4 {
5     reference_to<Simple_window>(pw).next();
6 }
```

- The `reference_to<Simple_window>(pw)` tells the compiler that the address in `pw` is to be treated as an address of a `Simple_window`
- From the reference, we call the member function `next()` – out of the system-dependent code as quick as possible
- Purpose of keeping `next()` separate from `cb_next()` was to keep system-dependent part separate from our user code
- `next()` function then performs the desired action

Review of steps to get to our `next()` function

- Steps to get our `next()` function called:
  1. Define our `Simple_window`
  2. `Simple_window` registers its `next_button` with GUI system
  3. We click the image of the `next_button` and GUI calls `cb_next()`
  4. `cb_next()` converts low-level system info into a call to our `next()` function for our window
  5. `next()` performs whatever action we want done in response to the button click
- Technique deals with all kinds of interactions – not just our button push
- Window can have many buttons, and program many windows
- Understanding how `next()` is called – we understand how to deal with every action in a program with GUI interface

A wait loop

- `next()` button in this case stops execution of program by setting `button_pushed` to true
- Ie. we had called:

```
win.wait_for_button();
```

- Which calls `fltk`'s version of `wait()` that takes care of everything until the "Next" button is pressed:

```
void Simple_window::wait_for_button()
// modified event loop:
// handle all events (as per default),
// quit when button_pushed becomes true
// this allows graphics without control inversion
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```

A lambda expression as a callback

- For each button we define two functions: one to map from system's notion of a callback, and one to do our desired action
- For example, we eliminate `cb_next()` and rewrite the constructor as:

```
1  Simple_window::Simple_window( Point xy,
2    int w, int h, cost string& title ) :
3    Window{ xy, w, h, title },
4    next_button{ Point{x_max()-70,0}, 70, 20, "Next",
5        [](Address,Address pw){
6          reference_to<Simple_window>(pw).next(); } },
7    button_pushed{false}
8  {
9    attach( next_button );
10 }
```

Button

- We define a `Button` as:

```
1 struct Button : Widget {
2     Button(Point xy, int w, int h, const string&
          label, Callback cb)
3         : Widget{ xy, w, h, label, cb} {}
4     void attach(Window&);
5 };
```

- The `Button` is a `Widget` with a location `xy`, a size `(w,y)` and a text label tt label
- A *Widget* is the technical term for a control.
- We use widgets to define the forms of interaction with a GUI

Widget

- Our `Widget` interface class looks like:

```
typedef void(*Callback)(Address, Address);
class Widget {
  public:
    Widget( Point xy, int w, int h,
            const string& s, Callback cb);
    virtual void move( int dx, int dy );
    virtual void hide();
    virtual void show();
    virtual void attach( Window& ) = 0;
    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;
  protected:
    Window* own;  // Window containing Widget
    Fl_Widget* pw;// FLTK Widget
};
```

Notes on `Widget`

- A `Widget` starts out visible, but can be made invisible, or visible again using the methods `hide()` and `show()`
- Like a `Shape` we can move a `Widget` using the `move()` method
- A `Widget` needs to be attached to a `Window` before it is used
- Note that `attach()` was declared as a "pure virtual" function
- That makes `Widget` a base class – representing a window control
- Each class derived from `Widget` must define its `attach()` method with what it means for it to be attached to `Window`
- `Widget::attach()` is what is called by `Window::attach()` as part of its implementation
- Result is that `Window` then knows about the `Window` and the `Widget` knows which `Window` it is in

Notes on `Widget`

- Definition of `Widget` and classes depending on it such as `Button, Menu`, etc. are in GUI.h
- Note that `Window` doesn't know what kind of `Widget` it deals with
- Similarly `Widget` doesn't know what kind of `Window` it deals with
- Note that some data members were left accessible
- The `Window * own` and `Fl_Widget * pw` members are strictly for the implementation, so are declared as `protected`

Notes on `Button`

- Recall the `Button` interface is declared as:

```
1 struct Button : Widget {
2     Button( Point xy, int w, int h,
3             const string& label, Callback cb)
4         : Widget(xy,w,h,label,cb){ }
5     void attach(Window&);
6 };
```

- the `attach()` function contains the relatively messy `fltk` code, which we won't look at for now
- note that placing a `Shape` over a button doesn't affect the button's s ability to function
- The shape of the button is left beyond our control, other than its location and size

In_box

- Two widgets are provided to get text from the GUI into and out of our program. Consider the `In_box` interface:

```
1 struct In_box : Widget {
2   In_box(Point xy, int w, int h, const string& s)
3     :Widget(xy,w,h,s,0) { }
4   int get_int();
5   string get_string();
6   void attach(Window& win);
7 };
```

- The `In_box` accepts text typed into it.
- We get the text as a string with `get_string()`
- Or we can get an integer using `get_int()`
- Note that you can get a float by using stringstream to parse the string

Out_box

- A widget for presenting some message a user is the `Out_box`, with interface:

```
1 struct Out_box : Widget {
2   Out_box(Point xy, int w, int h, const string& s)
3     :Widget(xy,w,h,s,0) { }
4   void put(int);
5   void put(const string&);
6   void attach(Window& win);
7 };
```

- Similar to `In_box` it provides methods for output of an int (`put(int)`) and for output of a string (`put( const string &)`)

Menu

- A widget for a vector of buttons as a `Menu` has interface:

```
1  struct Menu : Widget {
2    enum Kind { horizontal, vertical };
3    Menu(Point xy, int w, int h, Kind kk, const
         string& label);
4    Vector_ref<Button> selection;
5    Kind k;
6    int offset;
7    int attach(Button& b);  // Menu does not delete
       &b
8    int attach(Button* p);  // Menu deletes p
9    void show(); // show all buttons
10   void hide(); // hide all buttons
11   void move(int dx, int dy); // move all buttons
12   void attach(Window& win); // attach all buttons
13 };
```

- The menu is simply a vector of buttons laid out in a line
- You provide the top left corner, a width and height to resize buttons as they are added to the menu
- Note this is not a “pop up” menu

Example

- Lets build a simple window that allows a user to display a sequence of lines (`Open_polyline`) specified by user input of coordinate pairs
- The window will look like:

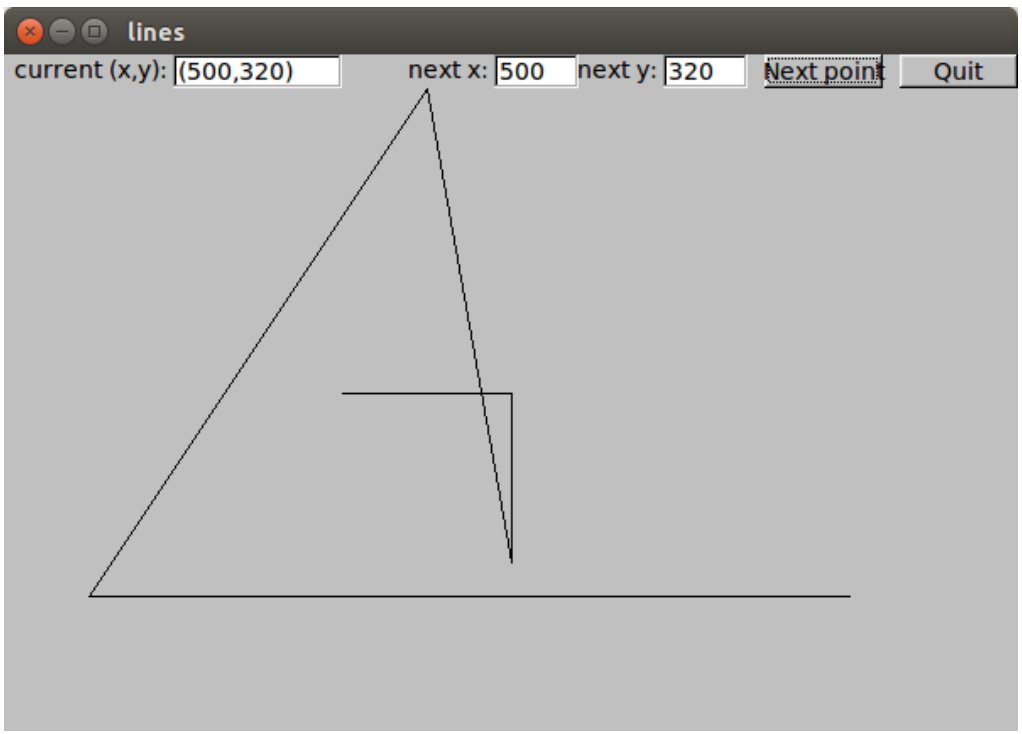


Figure: First example of using Widgets

Example

- Initially “current(x,y)” box is empty, and program waits for user to enter coord pair
- User enters values in “next x”, “next y” boxes and clicks “Next point” button to add the point
- Once done starting point appears in the “current(x,y)” box
- Each new coordinate pair entered draws a line from “current(x,y)” to the pair entered, and updates “current(x,y)”
- When the user is done they click a “Quit” button.

Class to represent window for drawing lines

```
1  struct Lines_window : Graph_lib::Window {
2    Lines_window( Point xy, int w, int h,
3                  const string& title );
4    Open_polyline lines;
5  private:
6    // add (next_x,next_y) to lines
7    Button next_button;
8    Button quit_button;
9    In_box next_x;
10   In_box next_y;
11   Out_box xy_out;
12   // callback for next_button
13   void next();
14   // callback for quit_button
15   void quit();
16 };
```

Notes on `Lines_window` class

- Line is represented by `Open_polyline`
- `Button`s and boxes are declared
- Each button has a member function implementing the desired action is defined `next()` and `quit()`
- Rather than having `cb_next()` and `cb_quit()` we use lambdas in the constructor.

Constructor for `Lines_window` class

```
1  Lines_window::Lines_window( Point xy, int w, int h,
2                              const string& title ) :
3    Window{xy, w, h, title},
4    next_button{ Point{x_max()-150,0}, 70, 20,
5      "Next point", [](Address, Address pw){
6          reference_to<Lines_window>(pw).next(); } },
7    quit_button{ Point{x_max()-70,0}, 70, 20, "Quit",
8       [](Address,Address pw){
9          reference_to<Lines_window>(pw).quit(); } },
10   next_x{ Point{x_max()-310,0}, 50, 20, "next x:"},
11   next_y{ Point{x_max()-210,0}, 50, 20, "next y:"},
12   xy_out{ Point(100,0), 100, 20, "current (x,y):"}
13 {
14     attach(next_button);
15     attach(quit_button);
16     attach(next_x);
17     attach(next_y);
18     attach(xy_out);
19     attach(lines);
20 }
```

Lines_window::quit() and ::next()

- "Quit" button deletes the window
- This is done by call to `fltk`'s `hide()` function to delete the window from view

```
1  void Lines_window::quit(){
2    hide();
3  }
```

- `next()` button does real work of reading coord pair, updating `Open_polyline`, and updating `Out_box`:

```
1  void Lines_window::next(){
2    int x = next_x.get_int();
3    int y = next_y.get_int();
4    lines.add(Point(x,y));
5    // update current position readout:
6    stringstream ss;
7    ss << '(' << x << ',' << y << ')';
8    xy_out.put(ss.str());
9    redraw();
10 }
```

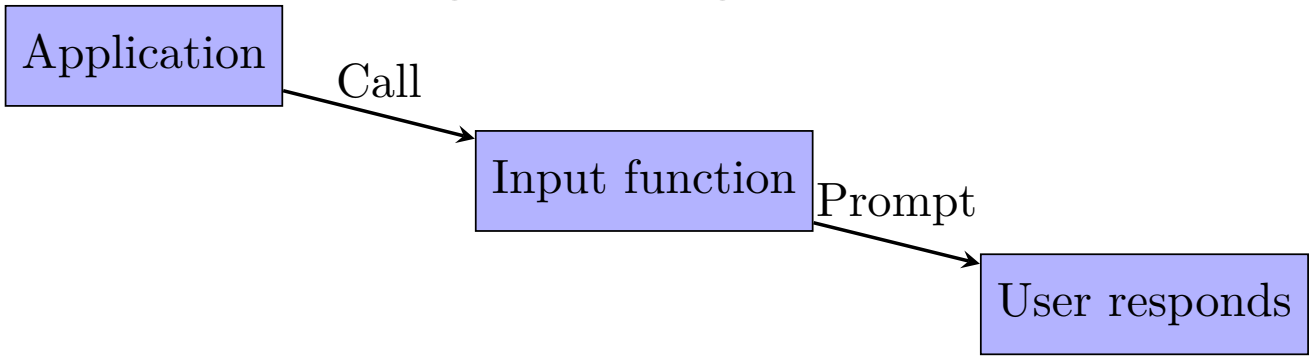
`main` function for `lineDrawing.cpp`

- Body of main is very short – we make our window and call `gui_main()` to hand control over to the `GUI`

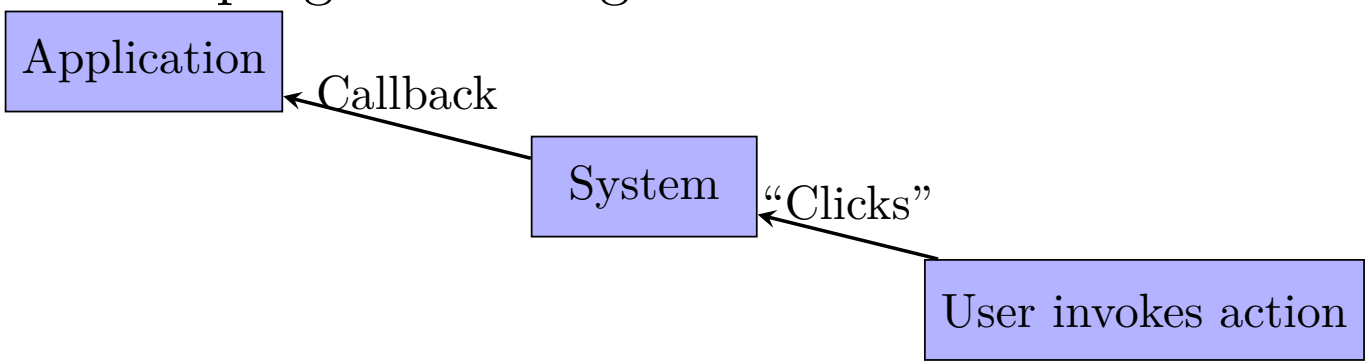
```
1  int main()
2  try {
3      Lines_window
           win(Point(100,100),600,400,"lines");
4      return gui_main();
5  }
6  catch(exception& e) {
7      cerr << "exception: " << e.what() << '\n';
8      return 1;
9  }
10 catch (...) {
11     cerr << "Some exception\n";
12     return 2;
13 }
```

Control inversion

- Conventional program is organized like this:



- A "GUI program" is organized like this:



Control inversion

- Implication of control inversion is that order of execution is determined by user interaction
- Makes systematic testing harder
- When coding GUI take extra care by making incremental improvements with testing in between
- Function invoked by a callback can do anything
- For now we'll just consider simplest case in which we hold data in our window

Adding a menu

- Lets provide a menu to change the color of all of the lines that are added:

```
1  struct Lines_window : Window {
2    //...
3    Menu color_menu;
4    static void cb_red( Address, Address);
5    static void cb_blue( Address, Address);
6    static void cb_black( Address, Address);
7    void red_pressed(){ change(Color::red); }
8    void blue_pressed(){ change(Color::blue); }
9    void black_pressed(){ change(Color::black); }
10   void change(Color c){ lines.set_color(c); }
11   //...
12 }
```

Initializing menu callbacks

- We also need to set up the callback in the constructor:

```
Line_window::Line_window( Point xy, int w, int h,
  const string& title ) :
  //...
  color_menu{ Point{x_max()-70,40},
    70, 20, Menu::vertical, "color" }
{
  //...
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "red", cb_red } );
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "blue", cb_blue } );
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "black", cb_black } );
  attach( color_menu );
}
```

- Buttons are attached to menu using `attach()`
- `Menu::attach()` sets size and location of buttons and attaches them to the window

Result of adding menu

Figure: `Lines_window` with color menu

Making our menu a pop open menu

- Lets add a “color menu” button that when pressed opens the color menu to choose one of the colors
- To do that we add another Button, and “pressed” functions to adjust visibility of that button and the color menu
- Here is what we add:

```
1  struct Lines_window : Window {
2    //... below all in private:
3    Button menu_button;
4    void hide_menu() {
5      color_menu.hide();  menu_button.show(); }
6    void menu_pressed() {
7      menu_button.hide(); color_menu.show(); }
8    void red_pressed() {
9      change(Color::red); hide_menu(); }
10   void blue_pressed() {
11     change(Color::blue); hide_menu(); }
12   void black_pressed() {
13     change(Color::black); hide_menu(); }
14   void menu_pressed() {
15     menu_button.hide(); color_menu.show(); }
16   //...
```

Updated constructor

```
1  Lines_window::Lines_window(Point xy, int w, int h,
       const string& title)
2    : Window{ xy, w, h, title},
3  //...
4    menu_button(Point(x_max()-80,30), 80, 20,
5       "color menu", cb_menu),
6  {
7    //...
8    attach(color_menu);
9    color_menu.hide();
10   attach(menu_button);
11   //...
12 }
```

- The initializers should be in the same order as the data member definitions
- Compilers will likely give a warning if they are out of order

Result of making popup menu

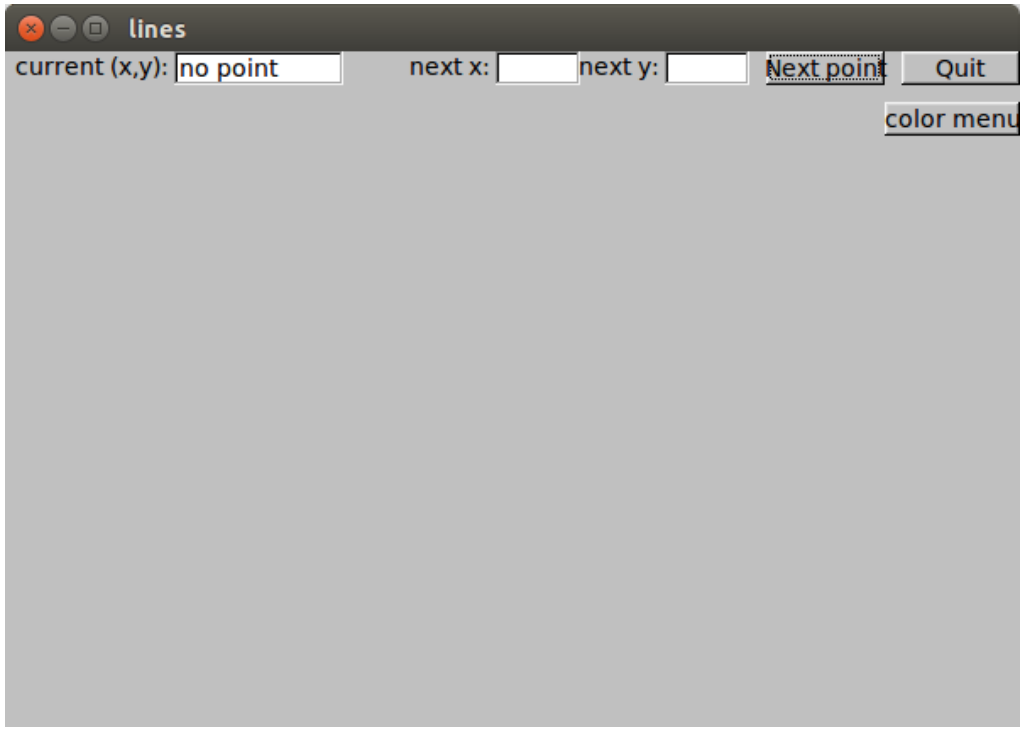


Figure: `Lines_window` with popup menu

Debugging GUI code

- Can be quite frustrating before first shapes and widgets start appearing in window
- Try this main():

```
1 int main(){
2   Lines_window{ Point{100,100}, 600, 400, "lines"
        };
3   return gui_main();
4 }
```

- Find the error! Try it even if you do spot the error.

Debugging GUI code

- How do you find errors in GUI programs?
  - Using well tried program parts
  - Simplifying all new code and slowly "growing" program
  - Carefully looking over the code line by line
  - Checking all linker settings
  - Comparing to code in an already working program
  - Explaining the code to a friend
- It is harder to trace execution of code – since it no longer goes in order
- Can also try using a “debugger” program
- Your code is not only one trying to interact with the screen

Debugging code

- Common problems to look for include:
  - Make sure object isn't hidden behind another object
  - Make sure `Shape` or `Widget` is attached, and that it doesn't go out of scope
- Most common problem is having object go out of scope after being attached, for example:

```
1  void disaster_fillmenu( Menu& m ){
2    Point zero{0,0};
3    Button b1{ zero, 0, 0, "flood", cb_flood );
4    Button b2{ zero, 0, 0, "fire", cb_fire );
5    m.attach( b1 );
6    m.attach( b2 );
7  }
8  int main(){
9    Menu disasters {Point{100,100},
10     60, 20, Menu::horizontal, "disasters" };
11   disaster_fillmenu( disasters );
12   win.attach( disasters };
13   return 0;
14 }
```

What is wrong with previous example code?

- All buttons were local to `disaster_fillmenu` function
- Attaching buttons to menu doesn't change scope of buttons
- One solution is to use `new` to allocate new memory for the objects – but then we lose the pointer to that location:

```
1 void disaster_fillmenu( Menu &m ){
2   Point zero{0,0};
3   m.attach( new Button{zero, 0, 0, "flood",
        cb_flood } );
4   m.attach( new Button{zero, 0, 0, "fire", cb_fire
        } );
5 }
```

- Another solution is to have a `Vector_ref< Button >` that is in a scope outside of `disaster_fillmenu` to hold the Buttons

Week 10 Done!

- Homework 7 is due Nov. 17 (17:00)
- Homework 8 is last homework is due Nov. 24 (17:00)
- Reminder: final project is due Dec. 1 (17:00)