Week 10 : Gaphing and GUI Classes

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 10

Outline

Outline

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

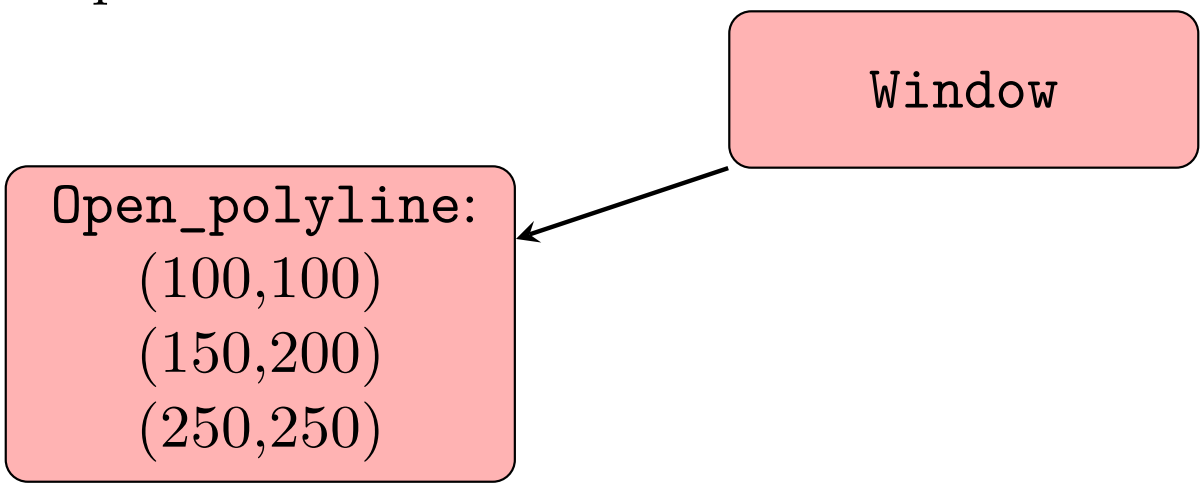
Naming of operations

- 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 }
```

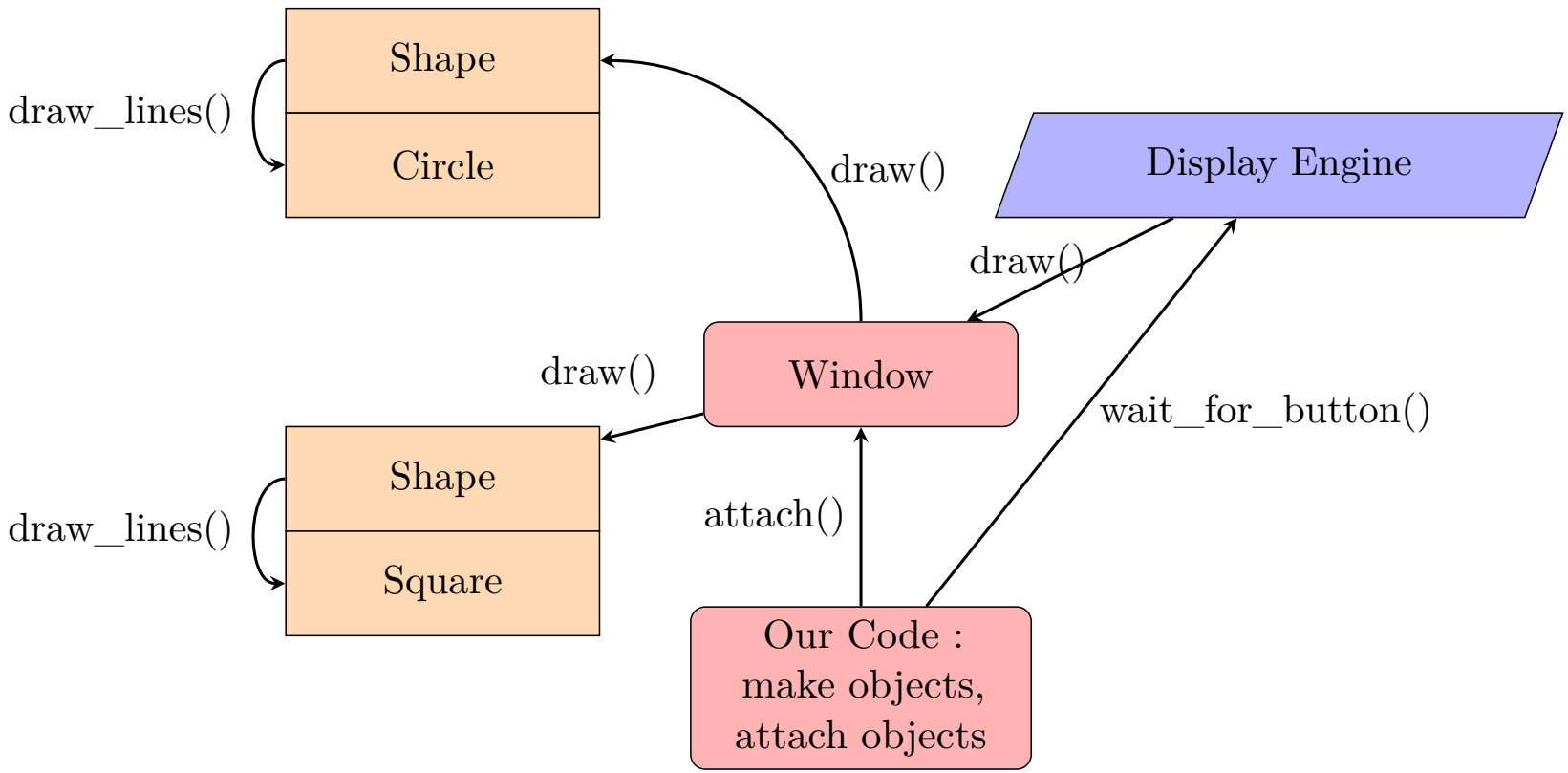
- `fltk` had no way to get current line style, so used `fl_line_style(0)` to reset to default at end
- `Shape::draw` does not change fill color or visibility – that is done by individual `draw_lines`

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;
```

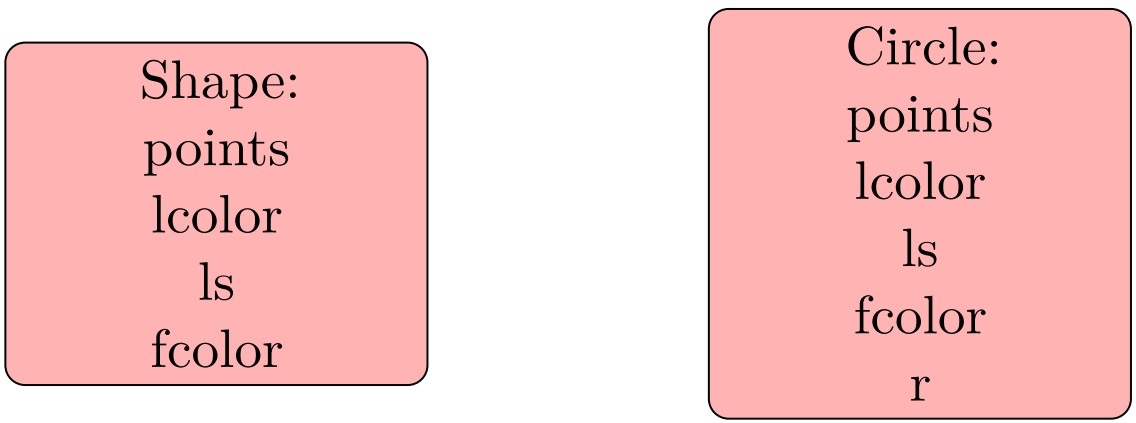
- Eliminates otherwise default copy operations, eg:

```
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



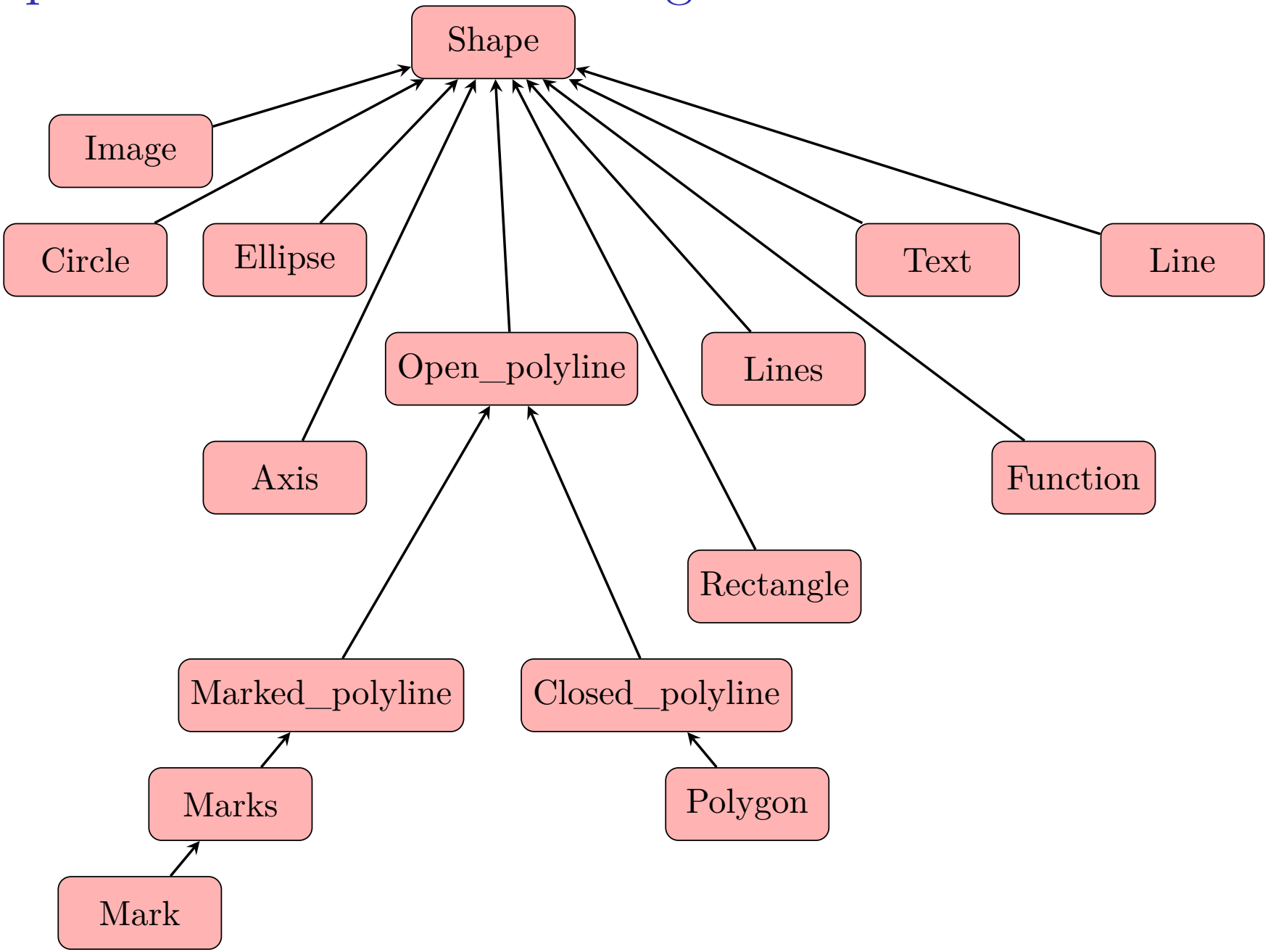
Copying and mutability

- 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

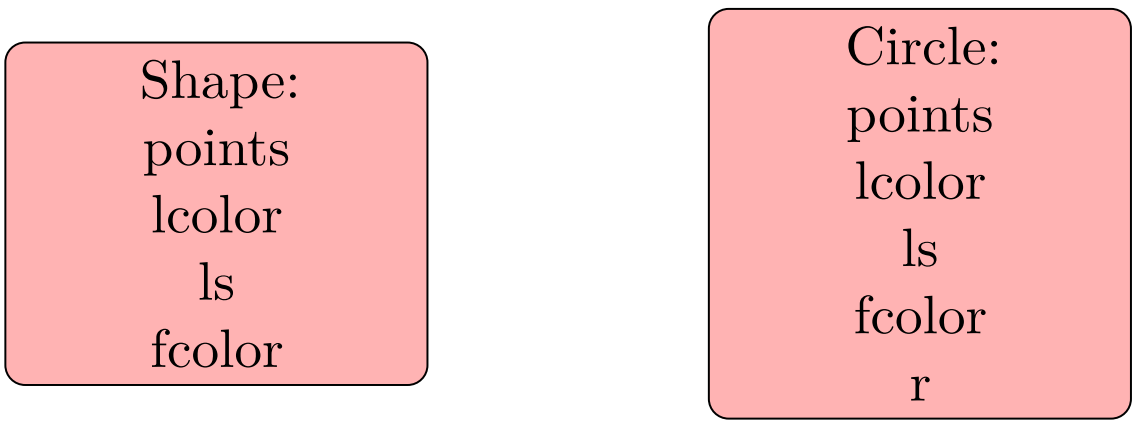
- In defining our graphics interface classes we used three language features:
  - *Derivation*: We build one class from another so that new class can be used in place of original. Ie. a `Circle` can be used in place of a `Shape`
  - *Virtual functions*: Ability to define function in base class and have 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

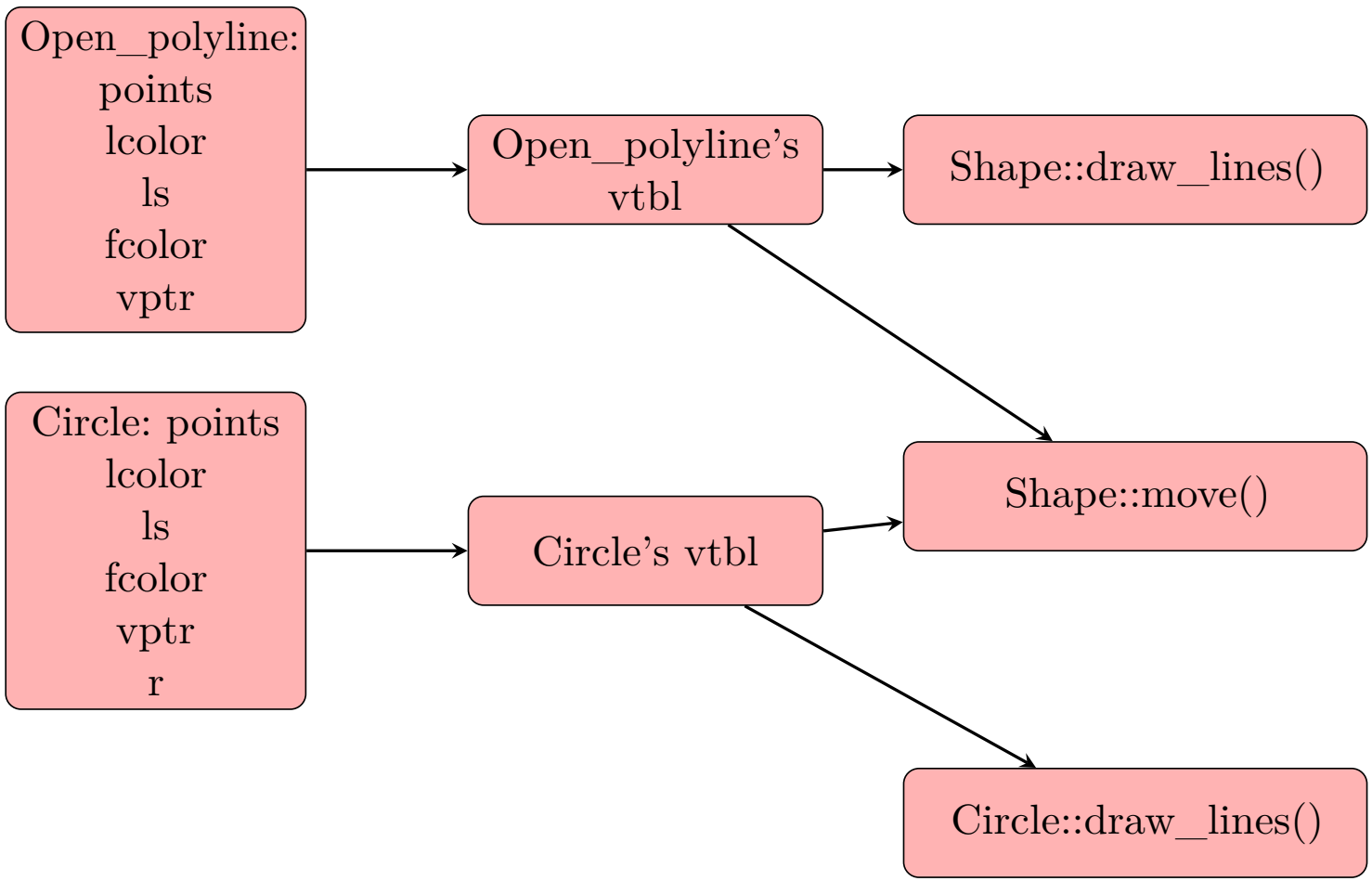
- Data members are stored one after another in memory
- Derived class's added data members get added after those in base class
- For example we saw layout of `Shape` and `Circle`:



- 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
```

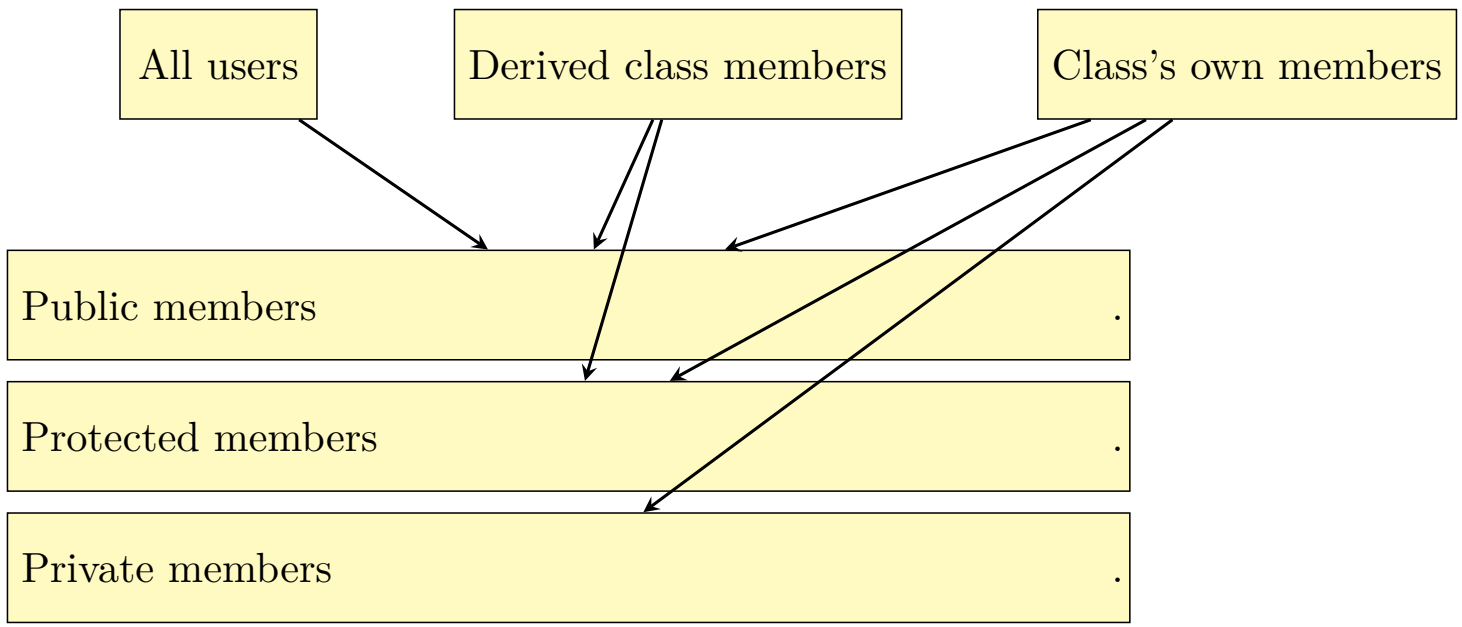
Overriding continuing

- 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:

```
1  class D1 : public B {
2    public:
3      // override virtual functions:
4      void f() override;
5      void g() override;
6  };
7  D1 d1; // Ok
8  class D2 : public B{
9    public:
10     void f() override;
11     // no g() ?
12 };
13 D2 d2;// error: D2 is (still) abstract
14 class D3 : public D2{
15   public:
16     void g() override;
17 };
18 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

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

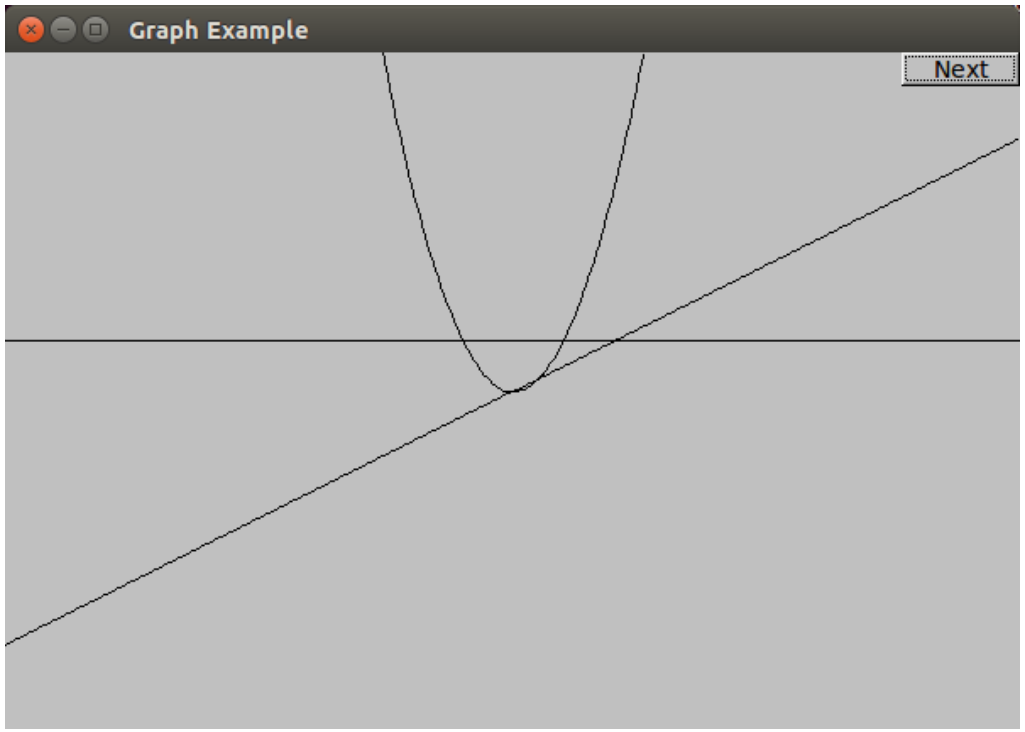


Figure : Result of `testGraph.exe`

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" };
```

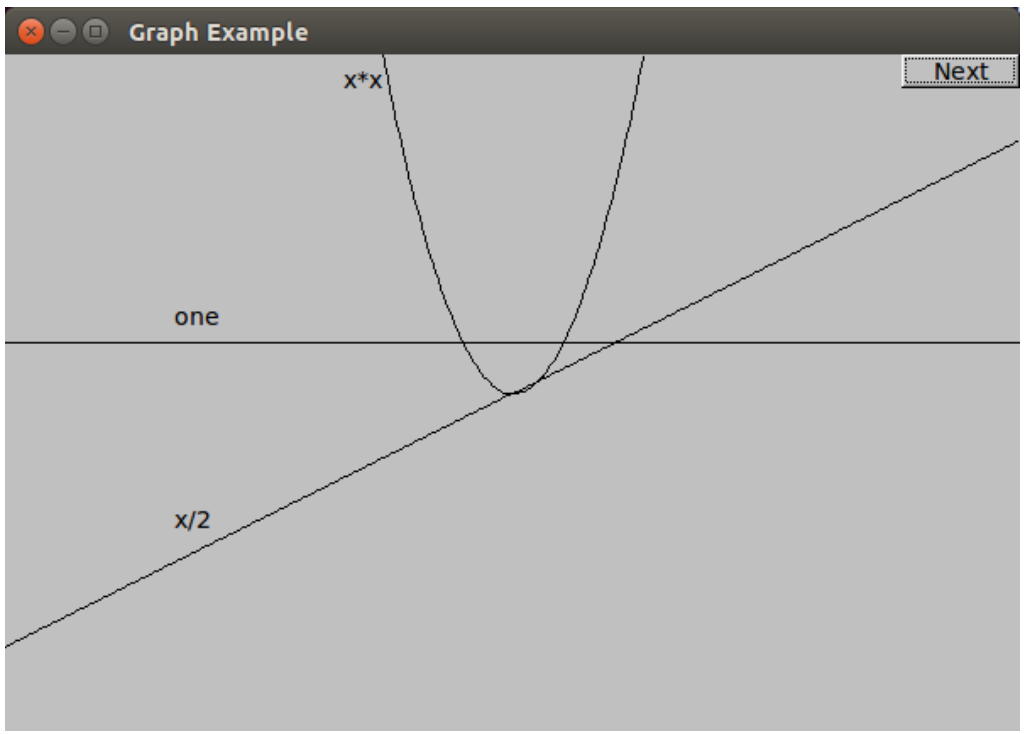


Figure : Updated result of `testGraph.exe`

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

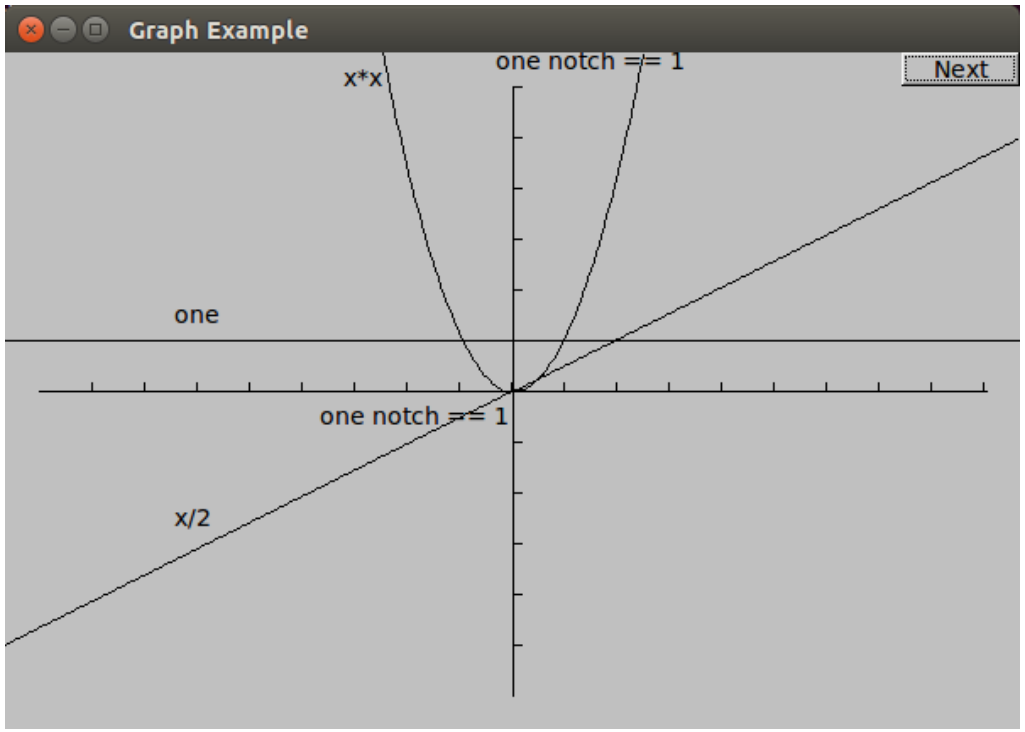


Figure : Updated result of `testGraph.exe`

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

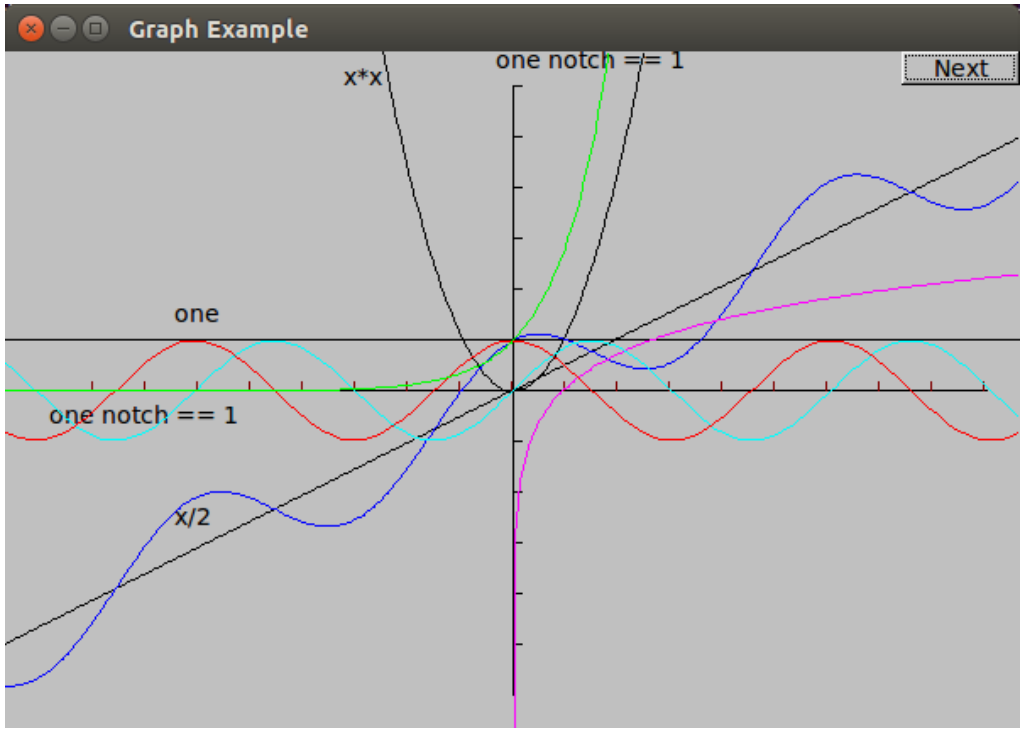


Figure : Updated result of `testGraph.exe`

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

Lambda expressions

- Defining a function just to pass it as an argument to our `Function` constructor can get tedious
- C++ has a notation for lambda expressions which define something that acts like a function
- The lambda expression for the sloping_cos function looks like one of the following two lines of code:

```
1 []( double x ){ return cos(x)+slope(x); }
2 [](double x) -> double { return cos(x)+slope(); }
```

- The second one explicitly states the return type, which is often not required
- The [] is called the *lambda* introducer, which is followed by the arguments to the lambda
- We could therefore define our `Function` as:

```
1 Function s4{ []( double x ){ return
      cos(x)+slope(x); },
2   r_min, r_max, orig, 400, 30, 30 };
```

`Axis` interface

- The interface for `Axis` is:

```
struct Axis : Shape {
  enum Orientation { x, y, z };
  Axis(Orientation d, Point xy, int length,
    int number_of_notches=0, string label = "");
  void draw_lines() const;
  void move(int dx, int dy);
  void set_color(Color c);
  Text label;
  Lines notches;
};
```

- Note that it has no private data
- Just a collection of semi-independent objects

`Axis` constructor

```
1  Axis::Axis(Orientation d, Point xy, int length, int n,
       string lab) : label(Point(0,0),lab) {
2    if (length<0) error("bad axis length");
3    switch (d){
4      case Axis::x: {
5        Shape::add(xy); // axis line
6        Shape::add(Point(xy.x+length,xy.y)); // axis line
7        if (1<n) {
8          int dist = length/n;
9          int x = xy.x+dist;
10         for (int i = 0; i<n; ++i) {
11           notches.add(Point(x,xy.y),
12                       Point(x,xy.y-5));
13           x += dist;
14         }
15       }
16       // label under the line
17       label.move(length/3,xy.y+20);
18       break;
19     }
20     // ...
```

`Axis` constructor cont...

```
    // ... continued
    case Axis::y: {
      Shape::add(xy); // a y-axis goes up
      Shape::add(Point(xy.x,xy.y-length));
      if (1<n) {
        int dist = length/n;
        int y = xy.y-dist;
        for (int i = 0; i<n; ++i) {
          notches.add(Point(xy.x,y),Point(xy.x+5,y));
          y -= dist;
        }
      }
      // label at top
      label.move(xy.x-10,xy.y-length-10);
      break;
    }
    case Axis::z:
      error("z axis not implemented");
  }
}
```

Notes on `Axis`

- Uses the `Shape` to store the line part of the Axis, by using `Shape::add()`
- `notches` are kept as a separate `Lines` object
- enumeration `Orientation` is used for the axis direction
- `Axis` has three parts to draw, so `draw_lines()` looks like:

```
1 void Axis::draw_lines() const {
2   // draw the line
3   Shape::draw_lines();
4   // the notches may have a different color from
        the line
5   notches.draw();
6   // the label may have a different color from the
        line
7   label.draw();
8 }
```

- Note the use of `draw()` for notches and label which first sets color, then calls `draw_lines()`, finally resetting the color

Notes on `Axis`

- Could set axis line, notches, and label to have different colors, or use method provided to set all three to same color:

```
void Axis::set_color(Color c) {
  Shape::set_color(c);
  notches.set_color(c);
  label.set_color(c);
}
```

- Similarly move all three parts of axis:

```
void Axis::move(int dx, int dy) {
  Shape::move(dx,dy);
  notches.move(dx,dy);
  label.move(dx,dy);
}
```

Introduction to approximations

- We will use graphics to illustrate approximating a function by Taylor series
- For example Taylor series of exponential function is:

$$\mathsf{e}^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ... \qquad (1)$$

- where ! is usual factorial (ie. $4! = 4 \times 3 \times 2 \times 1$)
- The more terms we add in this sequence the more precise our value of e^x
- We can compare graphs of functions:

```
exp0(x) = 0; // no terms
exp1(x) = 1; // one term
exp2(x) = 1+x; // two terms
exp3(x) = 1+x+pow(x,2)/factorial(2);
exp4(x) = exp3() + pow(x,3)/factorial(3);
...
```

Functions to solve our approximation problem

- Need a factorial function (not in std library):

```
int fac( int n ){
  int r = 1;
  while (n>1){
    r*=n;
    --n;
  }
  return r;
}
```

- We also need nth term in series:

```
double termn( const double x, const int n ){
  return pow( x, n ) / fac( n );
}
```

Functions to solve our approximation problem

- Now exponential up to term n is:

```
double expo( const double x, const int n ){
  double sum = 0;
  for ( int i=0; i<n; ++i ) sum += termn( x, i );
  return sum;
}
```

- We can define the `Function` that uses standard library exponential to compare to:

```
Function realexp{ exp, r_min, r_max, orig, 200,
    x_scale, y_scale };
realexp.set_color( Color::blue );
```

- But how to use our function `expo` that takes two arguments?

Using lambda to adapt our function

- We can use a lambda expression to wrap our function, so our code to successively display better and better approximation is:

```
for ( int n = 0 ;  n < 50 ; ++n ){
  ostringstream ss;
  ss << "Exp approximation to n=" << n;
  win.set_label( ss.str() );
  Function e{ [n](double x){ return expo(x,n); },
    r_min, r_max, orig, 200, x_scale, y_scale };
  win.attach(e);
  win.wait_for_button();
  win.detach(e);
}
```

- lambda inducer `[n]` says lambda expression may access local variable n
- For this to work however, need to use std::function from `<functional>` header and make an overloaded version of `Function`... not so simple

My solution to adapt our function

- We can use a static variable to keep track of n

```
1  // wrapper for expo to store n for next call to
       expo
2  double expo1( const double x, const bool
       change=false, const int nn=1 ){
3    static int n = 1;
4    if ( change ) n = nn;
5    return expo( x, n );
6  }
7  // wrapper for expo1 to make single parameter
       approx
8  // to exponential(x)
9  double expo2( const double x ){
10   return expo1( x );
11 }
```

- Call `expo1( x, true, n)` to change value of n
- Call `expo2(x)` as single parameter function

approxExpo.cpp

```
for ( int n = 0 ;  n < 50 ; ++n ){
  ostringstream ss;
  ss << "Exp approximation to n=" << n;
  win.set_label( ss.str() );
  expo1( 1.0, true, n );
  Function e{ expo2, r_min, r_max, orig,
              200, x_scale, y_scale };
  win.attach(e);
  win.wait_for_button();
  win.detach(e);
}
```

- `win.detach` makes sure next iteration in loop doesn't have `win` try to draw Function that went out of scope

Result of `approxExpo.cpp` for $n = 0$

- Note that there appears to be some glitch in the drawing
- Likely an integer over-run somewhere due to large values of exponential

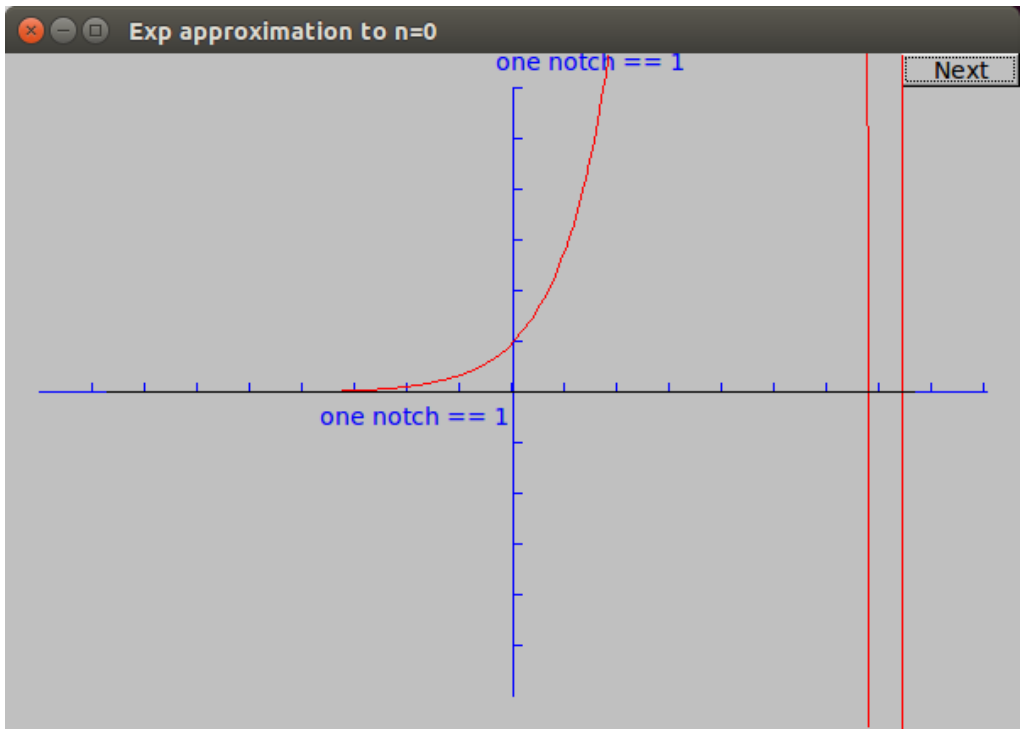


Figure : For $n = 0$ have straight line along $y == 0$

Result of `approxExpo.cpp` for $n = 1$

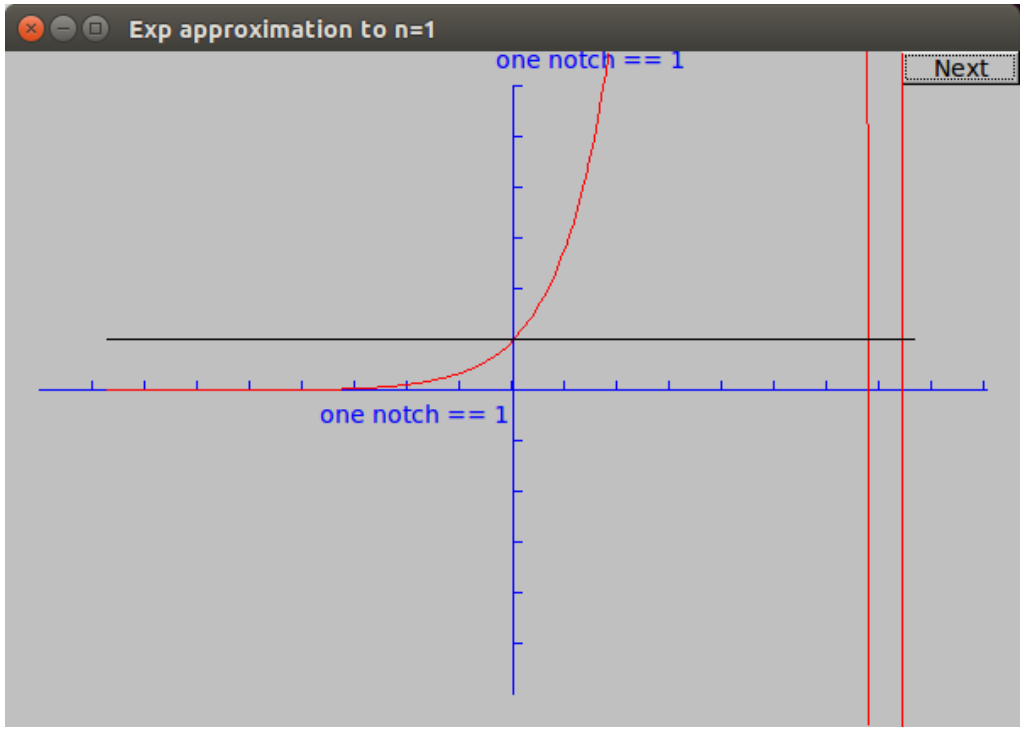


Figure : For $n = 1$ have straight line $y = 1$

Result of `approxExpo.cpp` for $n = 2$

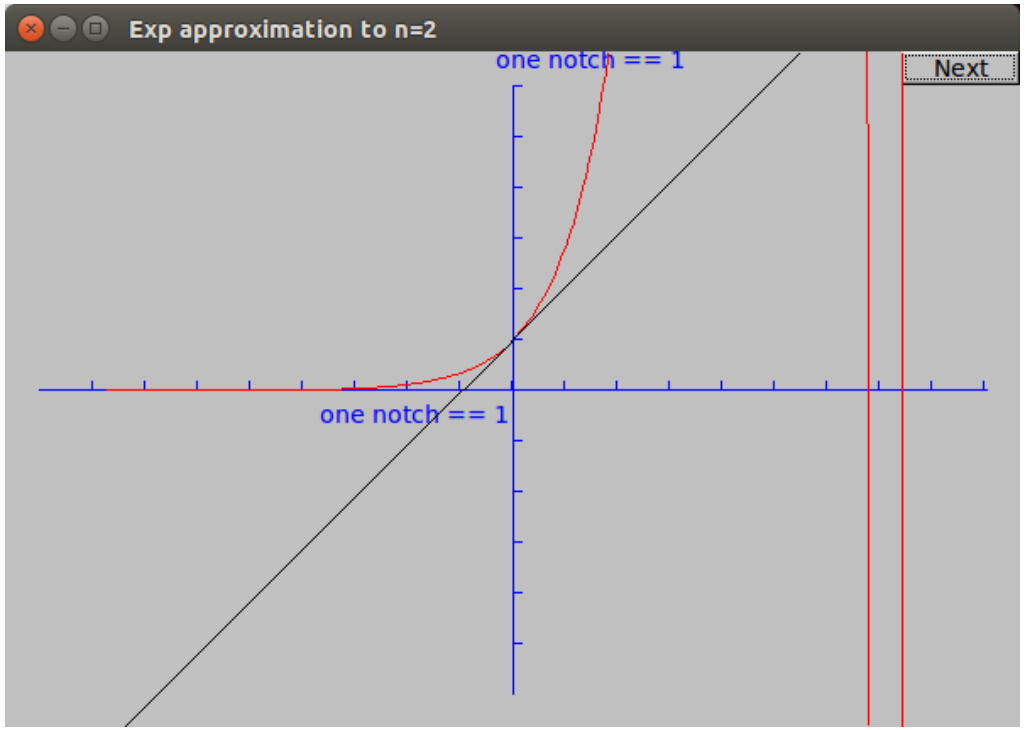


Figure : For $n = 2$ have slope line along $y = 1 + x$

Result of `approxExpo.cpp` for $n = 3$

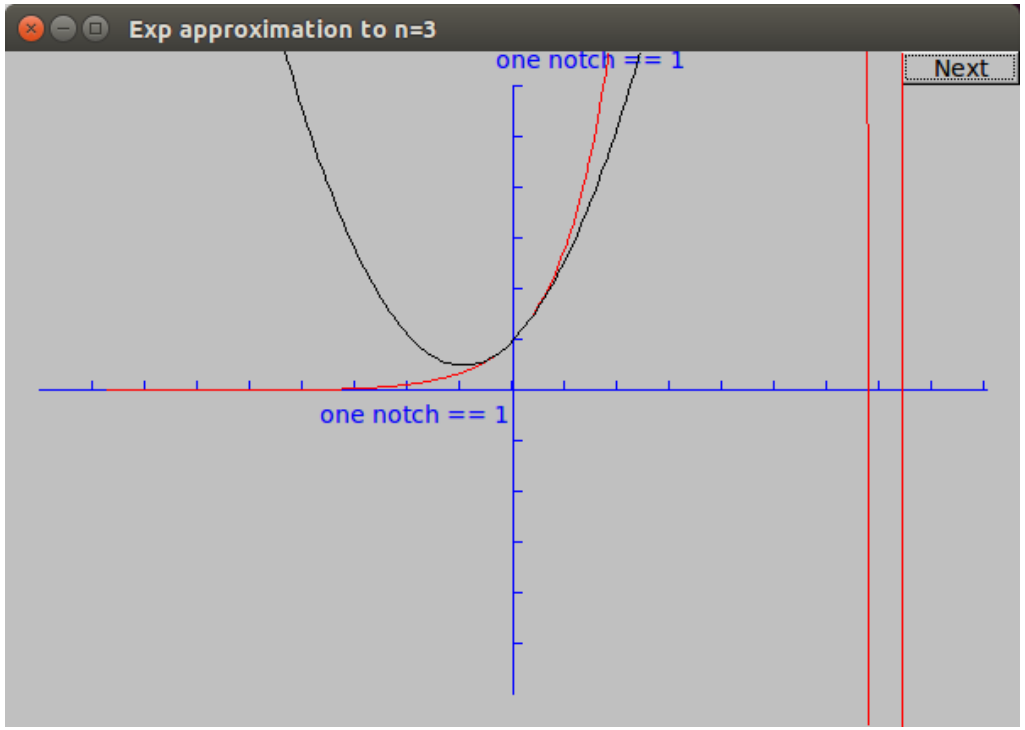


Figure : For $n = 3$ have parabola along $y = 1 + x + x^2$

Result of `approxExpo.cpp` for $n = 10$

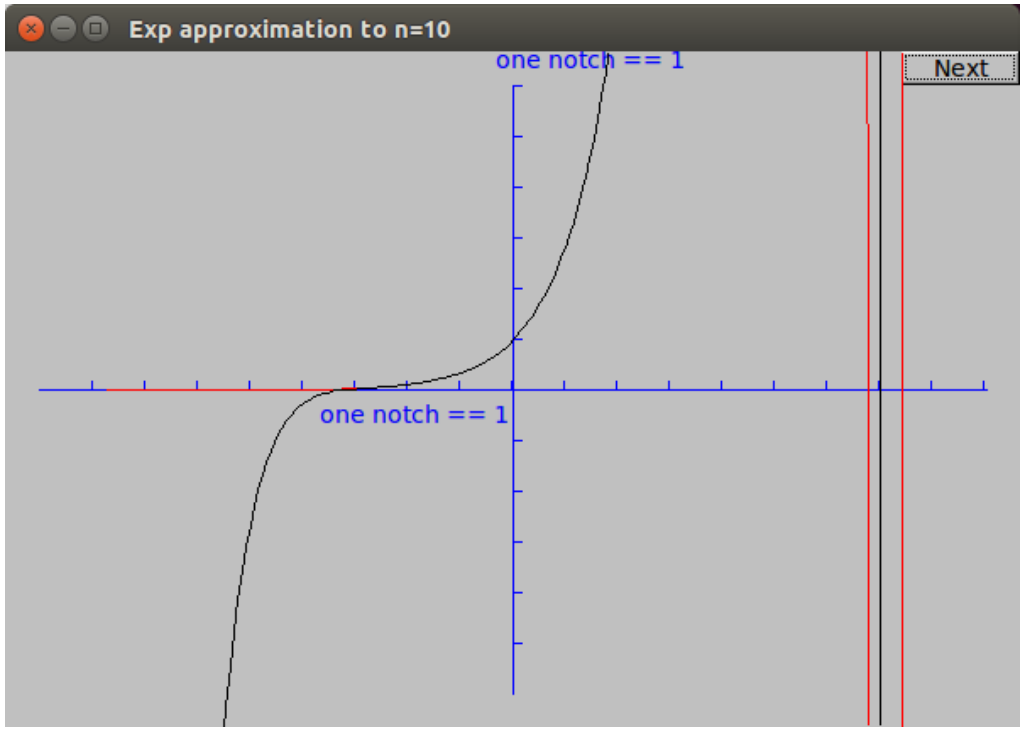


Figure : For $n = 10$ approximation is pretty good above $x = -3$

Result of `approxExpo.cpp` for $n = 18$

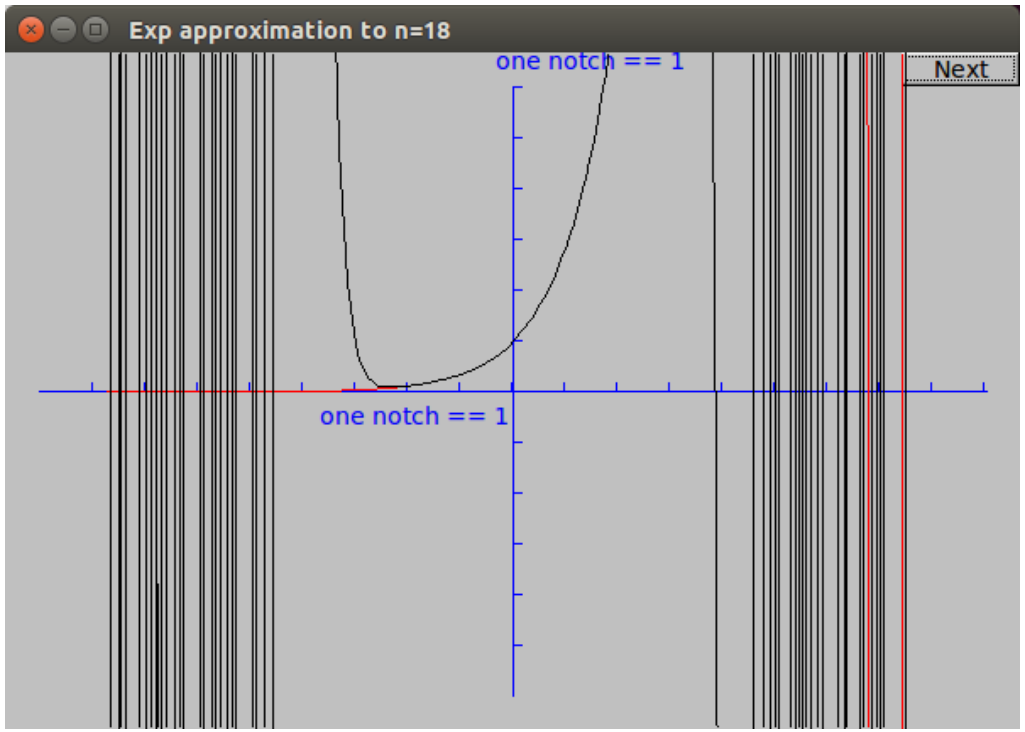


Figure : For $n = 18$ seems worse than previous iterations

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

Introduction to graphing data

- Graphing data is a highly valued skill
- Doing it well is as much an art as science
- Our simple plots will not reach a publication quality
- We will work towards a plot of the age of Japanese people and projection into future

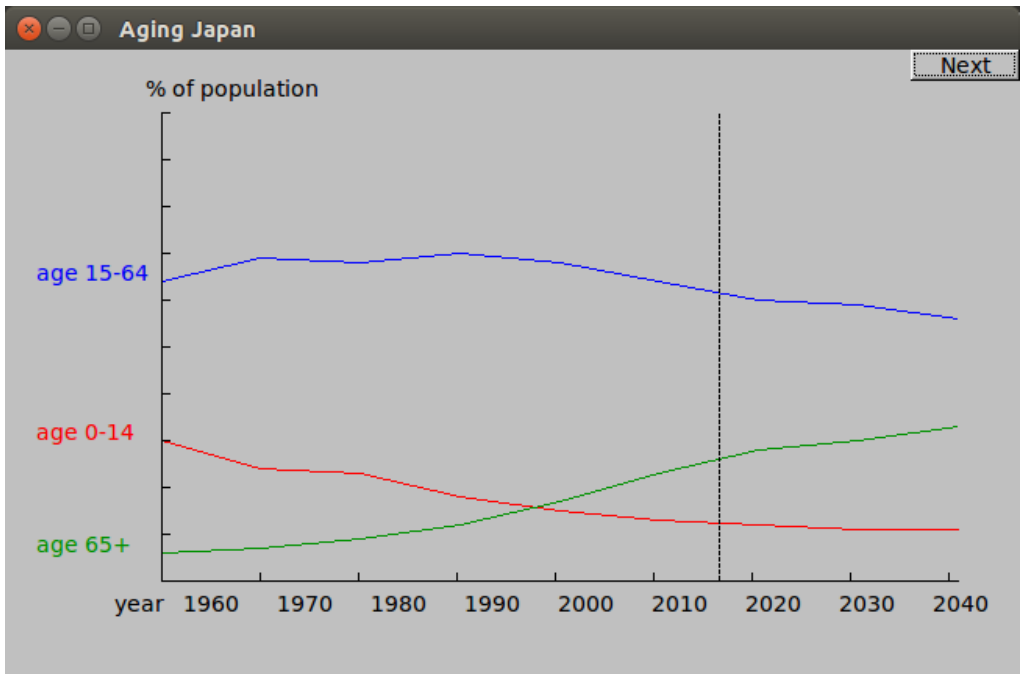


Figure : Graph that we will make

Parts needed for graphing data

- Reading a file of data
- Scaling data to fit the window
- Displaying the data
- Labelling the graph

Reading the file

- The data we are interested in plotting is in lines of the form:

```
( 1960 : 30 64 6 )
(1970 : 24 69 7 )
(1980 : 23 68 9 )
```

- Format gives year, followed by percent aged 0-14, aged 15-64, then aged 65+
- To keep code organized, often is useful to define a struct or class to read in data that is in a peculiar format

Reading the file

- Since our data is a type of distribution of ages, lets call the structure `Distribution`:

```
1  struct Distribution {
2    int year, young, middle, old;
3  };
4  // assume format: ( year : young middle old )
5  istream& operator>>(istream& is, Distribution& d){
6    char ch1 = 0; char ch2 = 0; char ch3 = 0;
7    Distribution dd;
8    if (is >> ch1 >> dd.year
9           >> ch2 >> dd.young >> dd.middle
10          >> dd.old >> ch3) {
11     if (ch1!= '(' || ch2!=':' || ch3!=')') {
12       is.clear(ios_base::failbit);
13       return is;
14     }
15   } else
16     return is;
17   d = dd;
18   return is;
19 }
```

Reading the file

- Use of `Distribution` struct splits code up into logical parts
- Wasn't strictly required – could have had one long sequential code read the data in then plot it
- But splitting the code into conceptual blocks helps make the code easier to debug
- Also makes it easier for someone else to read

Reading the file – read loop

```
string file_name = "japanese-age-data.txt";
ifstream ifs(file_name.c_str());
if (!ifs) error("can't open ",file_name);
//...
Distribution d;
while (ifs>>d) {
  if (d.year<base_year || end_year<d.year)
    error("year out of range");
  if (d.young+d.middle+d.old != 100)
    error("percentages don't add up");
  //...
}
```

Reading the file – notes

- Here we hard coded the filename into the program
- That's usually not a good idea for a long lived program
- Sometimes the code is meant to be a quick one off
- For longer lived code add extra code needed to make it more general
- Note that we check that the year is in some range
- Also check that percentages add up to 100%

Layout for the graph

- We will use `Open_polylines` to represent the data – one for each age group
- Need to label datasets – use colour and label somewhere along line
- Label x-axis with years
- Vertical line through 2016 to indicate the current year
- Getting placement of components of graph correct is fairly tricky and requires calculating several sizes and offsets
- Often a good idea to define several constants to help with various locations on the screen
- A quick sketch of the locations can help too!

A picture of the layout

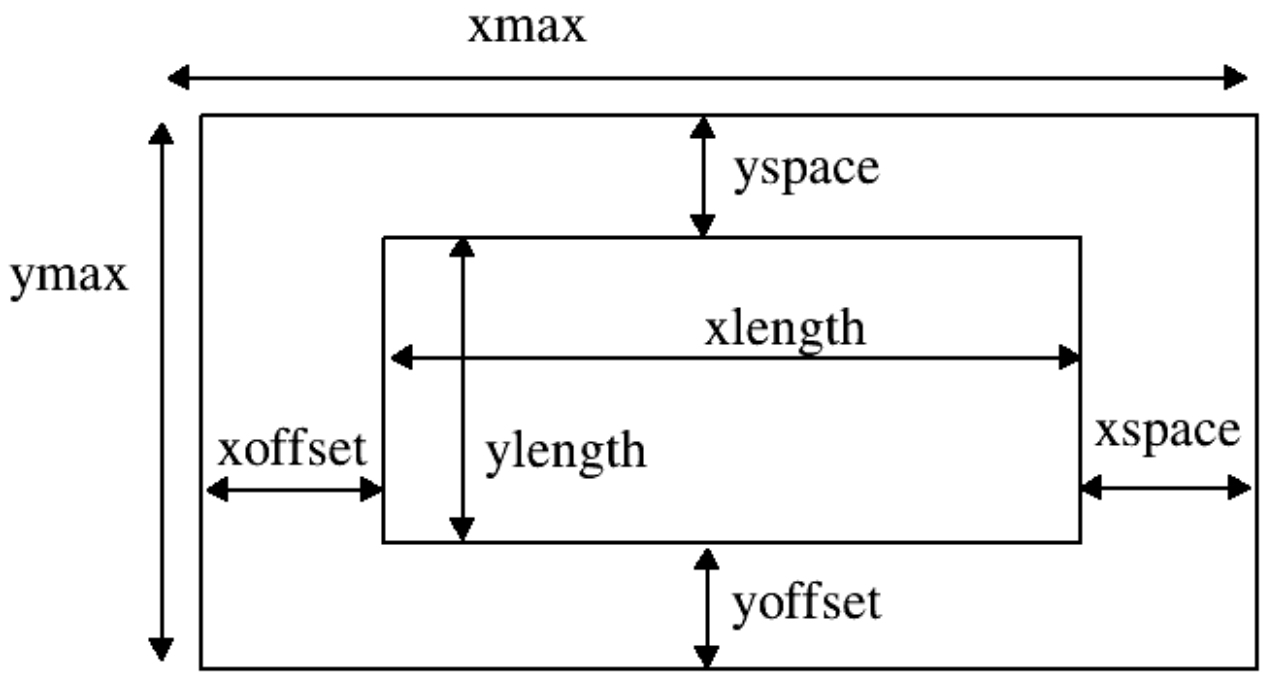


Figure : Sketch of offsets to locations on screen

Definitions of `const` values for layout

```
1  constexpr int xmax = 650; //window size
2  constexpr int ymax = 400;
3  // distance from left hand side of window to y axis
4  constexpr int xoffset = 100;
5  // distance from bottom of window to x axis
6  constexpr int yoffset = 60;
7  constexpr int xspace = 40;   // space beyond axis
8  constexpr int yspace = 40;
9  // length of axes
10 constexpr int xlength = xmax-xoffset-xspace;
11 constexpr int ylength = ymax-yoffset-yspace;
```

Scaling the data

```
constexpr int base_year = 1960;
constexpr int end_year  = 2040;
// scale of x values
constexpr double xscale =
  double(xlength) / (end_year-base_year);
// scale of y values
constexpr double yscale=double(ylength)/100;
```

- Converted `int` values into `double` to avoid rounding errors on scales for axes

`class` to do Scaling

- To scale a year onto screen, we subtract xoffset then scale by xscale:

```
1  // data value to coordinate conversion
2  class Scale {
3    int cbase;     // coordinate base
4    int vbase;     // base of values
5    double scale;
6  public:
7    Scale(int b, int vb, double s)
8      : cbase(b), vbase(vb), scale(s) { }
9    int operator()(int v) const {
10     return cbase + (v-vbase)*scale;
11   }
12 };
```

The scaling factor

- We define the scaling factors as:
  `Scale( int coordBase, int baseValue, doube scale )`

```
1 Scale xs{ xoffset, base_year, xscale };
2 Scale ys{ ymax-yoffset, 0, -yscale};
```

- `yscale` was made negative since the coordinates grow downwards, whereas graph increases upwards
- Then when we want coordinates on screen for `value` object `xs`:

```
int locx = xs( value );
```

- `locx` will then have result of `xoffset + (value-base_year)*xscale`

Building the graph

- Now we can build the graphs axes and line at 2016 as follows:

```
Simple_window win(Point{100,100}, xmax, ymax,
  "Aging Japan");
Axis x(Axis::x, Point{xoffset,ymax-yoffset},
  xlength, (end_year-base_year)/10,
  "year   1960      1970      1980      1990      "
  "2000      2010      2020      2030      2040");
x.label.move(-100,0);
Axis y( Axis::y, Point{xoffset,ymax-yoffset},
    ylength, 10,"% of population");
Line current_year( Point{xs(2016), ys(0)},
    Point{xs(2016), ys(100)});
current_year.set_style(Line_style::dash);
```

- Axes cross at `(1960,0)` at `xoffset, ymax-yoffset` on window

Notes on our axes

- Notches on x-axis are calculated to be every 10 years based on constants that where defined – if the data is updated this can easily be extended
- Notches on y-axis are every 10%
- Axis label string is a bit curious, two adjacent string literals were used:

```
"year    1960        1970        1980        1990        "
"2000        2010        2020        2030        2040"
```

- Adjacent string literals are concatenated by the compiler
- The label string is not very nice, since it had to be adjusted by hand to put the numbers next to the tick marks
- To do better we would have to build set of labels – one for each tick mark then build string for overall label
- Placing line at 2016 simplified by using scaling class

Open_polylines for the data

```
1  Open_polyline children;
2  Open_polyline adults;
3  Open_polyline aged;
4  Distribution d;
5  while ( ifs >> d ) {
6    if ( d.year<base_year || end_year<d.year )
7      error("year out of range");
8    if ( d.young + d.middle + d.old != 100 )
9      error("percentages don't add up");
10   int x = xs( d.year );
11   children.add( Point{ x, ys( d.young) } );
12   adults.add( Point{x, ys( d.middle ) } );
13   aged.add( Point(x, ys( d.old ) } );
14 }
```

Notes on `Open_polylines` for the data

- Using the `Scale` objects `xs` and `ys` make adding points to our polylines trivial
- Little classes like `Scale` are great for simplifying notation and avoiding having scaling done in multiple places
- Reduces possibility of making an error – since we only need to write the scaling once
- Increases readability of code

Adding labels to the data sets

- Now we add colored text to label the datasets:

```
Text children_label( Point{20,children.point(0).y},
    "age 0-14");
children.set_color(Color::red);
children_label.set_color(Color::red);
Text adults_label( Point{20,adults.point(0).y},
    "age 15-64");
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);
Text aged_label( Point{20,aged.point(0).y},
    "age 65+");
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

Attaching objects to the window

- Finally, remember to attach all the `Shape` objects to the window and start the GUI

```
1     win.attach(children);
2     win.attach(adults);
3     win.attach(aged);
4     win.attach(children_label);
5     win.attach(adults_label);
6     win.attach(aged_label);
7     win.attach(x);
8     win.attach(y);
9     win.attach(current_year);
10    win.wait_for_button();
```

Result of our efforts

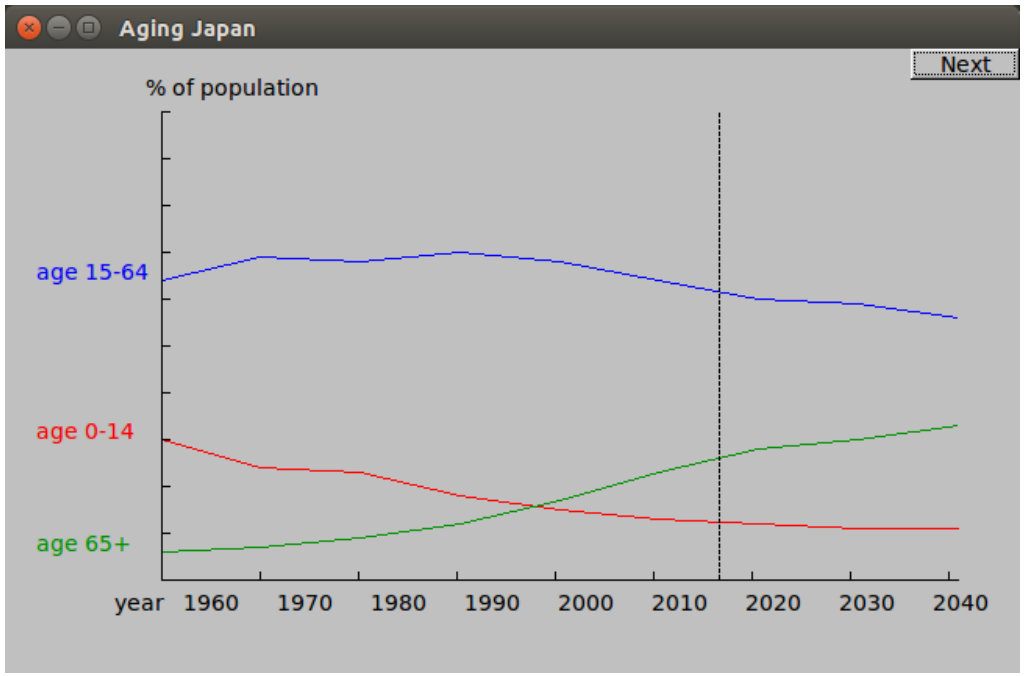


Figure : In case you forgot what we were plotting!

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

Week 10 Done!

- Homework 7 is due Nov. 18 (17:00)
- Homework 8 is last homework is due Nov. 25 (17:00)
- Reminder: final project is due Nov. 29 (17:00)